Language support for service-level agreements for application-service provision

James Skene

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

November 2, 2007

Declaration

I, James William Skene, confirm that the work presented in this dissertation is my own. Where information has been derived from other sources, I confirm that this has been indicated in the dissertation.

Legal disclaimer

Service-Level Agreements (SLAs) are discussed in this dissertation. Sometimes SLAs are legally enforcable contracts. In the British legal jurisdiction at least, it is illegal for unqualified persons to provide legal advice. Therefore the discussion of SLAs in this dissertation should be regarded as presenting the findings of research, and not legal advice. Nor should any part of this dissertation be regarded as legal advice. Any party seeking to enter a legally binding SLA should take advice from a qualified specialist lawyer before doing so. No representation is made that the SLAs described in this work are an appropriate basis for a legally binding SLA.

Abstract

My thesis is that practical language support can be provided for Service-Level Agreements (SLAs) for Application-Service Provision (ASP), which is better than that provided by pre-existing languages in that: it provides greater assistance in expressing conditions that mitigate the risks inherent in ASP; and disputes related to agreements expressed in this manner may be more easily resolved in so as to respect the original intent of the parties.

I support this thesis by establishing requirements for SLAs for ASP based on an account of a typical ASP infrastructure and business model. These identify the particular risks inherent in ASP, permit comparisons between ASP SLA languages, and guide the development of an abstract, extensible, domain-specific language, SLAng.

SLAng is defined using a meta-modelling approach that allows a high degree of precision in the specification of its semantics, traceability from SLA to language specification, and the testing of the language and SLAs to ensure they capture the original intent of the parties.

SLAng supports the expression of mutually-monitorable SLAs, for which the determination of compliance depends only on events visible to both client and provider of the service. I demonstrate that such SLAs are the most monitorable possible in a typical ASP scenario, given current monitoring technology, and describe an approximately-monitorable constraint on the accuracy of evidence used to administer such SLAs.

SLAng is shown to be of practical use in a case study, evaluated against the original requirements, and compared with pre-existing languages. The evaluation of SLAng is enhanced using metrics developed to assist in assessing the contribution of a domain-specific language specification to encoding the meaning of statements in that language.

Acknowledgements

Many people have provided me with assistance, support and feedback during the several years it has taken me to prepare this document and complete the research that it describes. Without doubt, the most important is Joanne Hacking, without whose love and encouragement I may not have endured the psychological rigours of studying for a PhD. I am also deeply indebted to my supervisor, Wolfgang Emmerich, for providing me with the opportunity and freedom to complete this work, and a broad education in academic life and standards. I have enjoyed working and publishing with D. Davide Lamanna, Giacomo Piccinelli, Jason Crampton, Franco Raimondi and my father, Allan Skene; special thanks must go to Jason and Franco for their efforts proof-reading this dissertation. Clovis Chapman and Liang Chen assisted me in the preparation of the case-study. My colleagues on the TAPAS, Divergent Grid and PLASTIC projects have contributed vital insights and influences to this work. I'd like to thank all my colleagues at UCL, for making the department a great place to be, and the pub an even better place to be. Finally, I must also thank my family as whole for their love and support, past, present and future; especially my parents and John Nickols, who together take full credit for raising me in a liberal and scientific tradition.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the outsourcing business model, a client organisation depends on one or more provider organisations to deliver services that realise some elements of the client's objectives. A reduction in the quality of these services causes some degree of suffering on the client's part, so the client needs to take measures to control its exposure to this risk, by ensuring that the services can be expected to be of a consistently high quality, or that the client is adequately compensated in the event of a deterioration of service quality. The stringency of such measures should of course be related to the value of the service to the client, and the client's perception of the likelihood of harm.

A number of measures can be taken by the client to control its exposure. They may select services on the basis of the reputation of a supplier, or on the maturity of the service, or on the degree of competition present in the marketplace for services, a possible driver of quality. The client may require that services must be implemented using particular technologies or methodologies, which guarantee certain properties of the services (e.g. hard-real-time operating systems offer the guarantee that correctly-implemented processes will complete within a fixed deadline [68]). The client may require a due-diligence inspection of the provider, to obtain some measure of confidence in the management of the service. The client may also or alternatively enter into a Service-Level Agreement (SLA) with the provider, in which constraints on the behaviour of the service are described, and financial penalties may be associated with violations of these constraints.

Application-Service Provision (ASP) is an umbrella term for the implementation of services in which a large component of the interaction between client and provider occurs over a computer network. Various middleware technologies have been designed to support ASP. When the client and provider are financially independent, ASP may be seen as an example of outsourcing, and is therefore expected to deliver the same benefits, allowing clients to concentrate on their core competencies and creating a competitive market for services, thereby lowering costs and driving quality. However, it has been asserted that the widespread adoption of this practice has been hindered by the high degree of diversity of technical services, and the lack of strong trust relationships in a global internet setting [97]. In this dissertation I argue that these factors represent financial risks to the client, and it is these risks that have limited the adoption of the ASP model. Clearly, SLAs are a potential mechanism to mitigate these risks, increasing the attractiveness of the ASP model.

1.2 Problem statement

The adoption of a technical language for specifying all or part of an SLA may be justified by a number of requirements for SLAs: most obviously by the common desire to utilise SLA information in technical service infrastructure, but also by the need to reduce the cost of SLA preparation without diminishing the quality of the SLAs. This cost may be reduced by the reuse of an appropriate language – the syntax to guide the design towards good SLAs, and the semantics to convey some of the intent of the agreement, reducing the effort required to author an SLA. Validation may be assisted by syntactic and semantic checking built into tools based on the language.

In all language design a tension exists between expressiveness and concision: the broader the domain of things that a language can describe, the more complex the language must become; or else the more abstract the concepts that it can express directly must be. In either case, the cost to the user of the language increases – either he must learn to use a more complicated language, or his statements must bear more of the burden of expressing his intent, and validation will be less automated. Therefore it is common to restrict the domain of a language to preserve concision. This decision, and the obvious appropriateness of SLAs to the ASP domain, appears to have motivated the design of a number of prior languages focussed on expressing SLA information that is applicable to the ASP domain.

None of these languages have found widespread adoption, and if the assumptions that that the ASP model is desirable, and that the use of appropriate SLAs makes it more desirable, are retained, then it is reasonable to conclude that these languages are not providing significant assistance in the production of appropriate SLAs. Having reviewed the prior languages, I contend that this is because none of the languages allow the expression of SLA conditions that convincingly mitigate the real risks involved in entering into an ASP relationship for at least one of two main reasons: either no support is provided for expressing the risk-mitigating conditions required, or the agreement once written seems to provide no real assurance that the parties can or will respect it. This latter flaw may be caused by two main deficiencies in an SLA: either a lack of precision in describing the agreement, or the inclusion of conditions that a dishonest or incompetent party could ignore without consequence. Note that if no confidence exists that parties will respect the initial intent of an SLA, then the SLA will not be effective as a means to mitigate risk, as a party may not be able to receive compensation for an injury it has sustained. Having the SLA would be no better than not having the SLA.

Clearly, the lack of language support for writing useful SLAs is a problem that is feasible to address and the solution to which will hopefully be of general benefit.

1.3 Contribution

The main contribution of this dissertation is to demonstrate convincingly the thesis stated in the front-matter, specifically that it is possible to provide practical language support for the authoring of ASP SLAs that is demonstrably better than that provided by previous languages in two particular ways: first, that real support is provided to express SLA conditions that mitigate the risks inherent to the ASP scenario; and second, that disputes concerning SLAs written in this manner will be easier to resolve in a manner consistent with the original intent of the agreement. This is achieved in three steps, described in the

following subsections:

1.3.1 Requirements analysis

First, the role of SLAs as a mechanism for mitigating risk in the ASP scenario is further motivated and a detailed list of requirements for such SLAs and a language in which to express them are developed. The requirements and the accompanying discussion of the ASP scenario establish the assumptions upon which this work rests. The requirements provide a basis for the comparison of ASP SLA languages, and inform the subsequent design of a new language. The requirements and the rationale behind them should be considered the first contribution of this work to ASP SLA design, since no previous work has presented a thorough treatment of requirements for ASP SLA languages.

1.3.2 The SLAng language

Second, an abstract core language of conditions for ASP SLAs, SLAng, is developed, in an attempt to satisfy the identified requirements to the maximum extent possible.

SLAng incorporates a number of theoretical advances, each of which represents a contribution of this work to the state of the art in ASP SLA language development, and in SLA development more generally. These are:

- **the adoption of the model-denotational approach to defining the abstract syntax and semantics of the language;**

  In this approach an object-oriented formalism is used to describe the structure and domain of a language. By applying this approach a high degree of precision and understandability in the definition of SLAng is achieved. The presence of the domain model and its explicit relationship to the structure of the language makes it straightforward to understand what aspects of the service are being constrained, what should be monitored, and how the parties should behave to comply with the SLA.

  I discuss the application of this approach to the problem of developing a language for SLAs, which suffers from conflicting requirements. An SLA language should reduce the cost of preparation of SLAs by encoding common domain knowledge, thereby allowing SLAs to be expressed concisely. However, the conditions required in an SLA may be related to a huge range of factors external to the technical implementation of a service, implying the need for a highly-expressive, generalised language. I describe how these requirements can be reconciled by the production of an abstract, extensible, domain-specific language, which captures the essential aspects of its domain, relies on the meta-modelling language in which it is defined to provide general expressive capabilities in extensions when required, but also provides structural guidance for the definition of those extensions.

  Based on experience obtained applying the approach, which was originally proposed by other researchers as a means to formalise modelling languages, I have suggested refinements to the underlying standards on which the approach relies, to improve its precision and to maintain traceability between statements and the languages in which they are written, general requirements inspired by the application of the approach to the SLA domain.

I have also demonstrated how generative programming standards can be combined with the approach to efficiently implement a checker component. This can be used as a syntactic and semantic checker for statements in the language. It can also be used to test the language, its extensions, and statements in the language, lending confidence to the assertion that an SLA written in the language genuinely captures the intent of the parties with respect to some agreement written in the language. I have also evaluated its use as part of the implementation of a runtime monitoring system for SLAs.

- **a method for the analysis of monitorability of systems of SLAs;**

  Monitorability concerns the ability for parties to obtain reliable evidence about the events pertinent to compliance with an SLA. A party entering into an SLA will have more confidence that disputes relating to the SLA will be resolved according to the original intent of the agreement if they can monitor compliance to the SLA by other parties to the agreement.

  Systems of SLAs may be classified according to the least monitorable SLA that they contain. The result of applying our analysis to a typical ASP scenario, involving financially independent client, service-provider and network-service provider parties, is that mutual monitorability is the best level of monitorability for a safe system of SLAs in the ASP scenario, assuming that tamper-proof monitoring systems are not available. Significantly, this degree of monitorability can only be achieved by a single configuration of SLAs in which parties only participate in SLAs which have conditions related to events occurring at the interfaces between their own technical infrastructure and that of another scenario participant. This result implies that network service providers may need to act as re-sellers of application services, a business model not in common usage today. It also suggests that only electronic-service oriented, rather than network-oriented SLA vocabulary, is necessary to insure end-to-end quality-of-service properties. Hence, this has allowed a focus on mutually monitorable, electronic-service oriented SLAs in the design of SLAng.

- **an approximately-monitorable measurement-accuracy constraint;**

  If SLAs are at best mutually monitorable, then the client and provider of the service will have to periodically meet to produce a reconciled account of service behaviour from which to calculate penalties. SLAng includes support for specifying how this procedure should take place; furthermore, the parties must be constrained to report honestly, whilst accommodating an inevitable amount of disagreement due to measurement error. I show how to write such a constraint in such a way that it is approximately monitorable using a statistical hypothesis test based on the comparison of trusted and un-trusted monitoring logs. Support for the constraint is included in the language.

- **support for expressing mutually-monitorable conditions appropriate to the scenario.**

  SLAng has been developed to include support for expressing conditions to mitigate risks implied by the scenario, in such a manner as to preserve mutual monitorability. These include constraints to mitigate risks due to bad behaviour by either party. Bad behaviour in general must be defined in

a service-oriented manner, but the prevalence of electronic services in ASP scenarios enables the need for reliability, latency and throughput conditions to be anticipated and supported. Conditions are also developed to mitigate the risk of early termination of the agreement by either party.

1.3.3 Evaluation

The final step in demonstrating the thesis is achieved by evaluating SLAng to show that it provides practical language support for ASP SLAs, can express SLAs that better mitigate the risks implied in the scenario than those expressible using prior languages, and that disputes concerning SLAs written using SLAng will be easier to resolve in a manner consistent with the original intent of the agreements, thereby using the example of SLAng to show that such support is possible.

The evaluation is achieved by a number of means:

- SLAng is evaluated critically in comparison to previous languages developed for the same purpose according to the criteria set by the requirements developed for such languages. The broad survey of ASP SLA languages and related technologies included in this evaluation is a contribution of this work to existing literature on SLAs;

- SLAng is used to support the expression of SLAs in a case study. The case study focusses on an existing service that allows the execution of large-scale computational experiments on grid resources, an endeavour involving several financially independent parties. I present a risk analysis of the scenario based on the activities required to conduct the experiment. I then design a system of SLAs capable of mitigating these risks, and implement the SLAs using extensions to SLAng;

- to assist in the evaluation of SLAng I develop a theory of metrics for domain-specific languages, based on the idea that the expressive burden of a statement is spread across the syntax of the statement and the definition of the language in which it is written. This idea gives rise to precise definitions of properties of languages with intuitive appeal: the power of a language in relation to a statement can be defined as the relative size of a statement and the language elements used to construct the statement; specificity and adequacy measures can be defined similarly. These metrics provide quantitative support for the qualitative argument that SLAng provides good practical support for expressing the SLAs required by the case study.

These exercises combine to demonstrate the thesis as follows: first, SLAng is shown to better express SLAs that mitigate the true scenario risks by identifying these risks in a theoretical discussion of the scenario, then by observing that these same risks credibly exist in the case-study scenario, and finally by finding in the critical review that other languages do not provide good support for mitigating these risks, whereas SLAng does.

Second, I argue that disputes concerning SLAs expressed in SLAng should be easier to resolve in a manner consistent with the original intent of the agreements by noting that SLAng benefits from a more precise language specification than alternative languages, and that its semantics support the creation of mutually-monitorable SLAs, whereas prior languages do not provide any explicit support for this. These two properties should tend to allow disputes to be more easily resolved in a manner consistent with

the original intent of the agreement, since the resolution of a dispute depends only on determining the intent of the SLA with regard to reliable evidence concerning the behaviour of the service. Also, by demonstrating tool support for checking the SLAs, I show that it is possible to obtain confidence that the SLAng SLAs capture the original intent, and that conformance to the SLA can be checked under some circumstances using the tools.

Finally the case-study has permitted a demonstration of the practicality of SLAng. I support this assessment quantitatively using my metrics to assess the power, adequacy and specificity of the core language and extensions. I demonstrate that evolution of the language will be possible in the future to increase its adequacy, with specificity measurements used to control the evolution, preventing from becoming bloated and therefore difficult to use.

1.4 Structure of the dissertation

In the next chapter, I introduce the ASP scenario and the use of SLAs in detail, in order to state the assumptions upon which this work is founded; I then develop requirements for systems of SLAs in the ASP scenario, languages for ASP SLAs and specification documents defining languages for ASP SLAs based on these assumptions.

In Chapter 3 I describe an approach to defining domain-specific languages appropriate to languages for ASP SLAs.

In Chapter 4 I describe enhancements to the underlying meta-model standards employed in Chapter 3 to improve the precision of language specifications and language statements. I also describe tools based on these standards and the proposed improvements, supporting the authoring an mechanical validation of language specifications and statements. I discuss the potential of such tools to assist in monitoring conformance to SLAs. Finally, I describe a theory of metrics that may be helpful in the evaluation and evolution of domain-specific languages.

In Chapter 5 I describe a theory of monitorability, and the results of a monitorability analysis applied to the ASP scenario; I also describe the design of a constraint on the accuracy with which parties must report measured values when administering an SLA, and demonstrate that the constraint is approximately monitorable.

In Chapter 6 I describe the design and implementation of the SLAng language.

In Chapter 7 I describe a case-study of the use of SLAng to specify SLAs for an application-service, provided by a federation of several financially-independent parties, implementing the facility to perform computation experiments of interest to chemists at University College London.

In Chapter 8 I summarise the evaluation of SLAng, as described above, including an evaluation of SLAng against my requirements, and in comparison to alternative languages. I also use the metrics developed in Chapter 4 to demonstrate the power, adequacy and specificity of the language in relation to the case-study, and to demonstrate a process of refinement by which the language may be improved in the future.

In Chapter 9 I summarise this work and discuss future research challenges.

Chapter 2

Requirements

In this chapter I describe the assumptions upon which this work rests, and then develop a set of requirements for systems of SLAs for ASP, languages for such systems, and the specifications of such languages, the quality of which has a direct impact on the practical usefulness of the language.

2.1 The Application Service Provision (ASP) scenario

In this section I start by examining the ASP scenario more closely.

In ASP, communication and processing are implemented to a large extent using electronic services. An electronic service is software executed on a network-connected node, and allows the communication with a client using protocols typically based on requests and responses.

At least three roles are usually involved in the provision of an electronic service. These are the client C, the service provider S and the network-service provider, in the context of the Internet also known as an Internet-Service Provider (ISP), denoted by I. The scenario is depicted in Figure 2.1.

The client, utilising some appropriate client software, submits requests to the service at its discretion, or according to a loose schedule. The network, under the supervision of the ISP conveys these requests to the service, which performs some appropriate processing, possibly performing or instigating some real-world activity as a result, and possibly storing or modifying some data held on behalf of the client. The performance of the service is the responsibility of the service provider. In due course, a response may be returned to the client via the network.

The implementation of this type of electronic communication is supported by a number of modern middleware systems, including various Remote Procedure Call (RPC) implementations [17, 128], the OMG's CORBA [90], Microsoft's DCOM [65] and .NET [66], Sun's J2EE [127], and Web Services [145]. Web-server technology, for example the Apache [4] web-server, also implements this type of communication, based on the HTTP protocol [37].

In any of the above mentioned technologies, requests to the service carry information in the form of parameters. In general, one parameter identifies the particular function of the service being invoked. I therefore state that a service consists of a set of named *operations*.

Responses may also convey information in the form of parameters or be empty signifying a simple acknowledgement of the request. I do not assume a synchronous model of communication between individual client programs and the server. Responses may never be generated. Multiple requests may be submitted by a single instance of client software before any response is returned by the service.

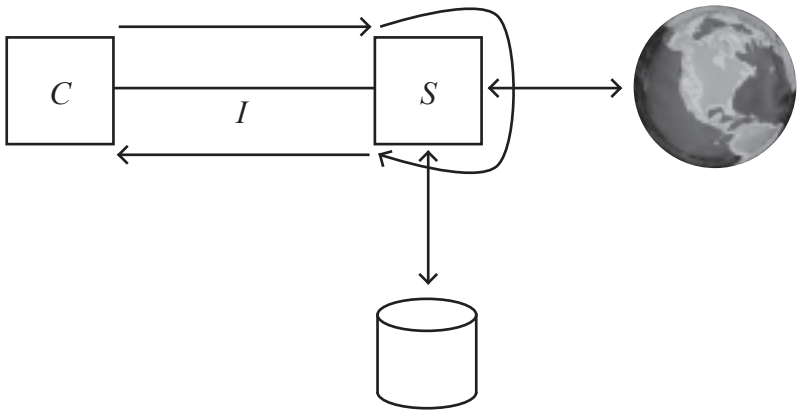


Figure 2.1: A three-party electronic-service scenario

The ASP scenario characteristically involves the provision of at least one electronic service by a provider party to a client party. However, multiple electronic services may be involved in the provision of one application service to a client. The service provider may permit the client to access multiple, related electronic services. The client software may also implement electronic services, as well as having the capability to access them. As part of the application service provided to the client, the service provider may spontaneously invoke operations on an electronic service implemented by the client software to push information to the client.

The distinction I maintain between electronic services and application services is that an application service consists in the overall delivery of some utility by the provider to the client, whereas electronic services are merely individual channels of communication. When I refer to client and provider parties in this work, I am referring to the client and provider of the application service, unless otherwise stated. I also refer to individual electronic services as service interfaces below, and use the term 'service' interchangeably to refer to either an overall application service or an individual electronic service, provided that it is clear from the context what is intended.

Commonly, more than one ISP may be involved in the delivery of messages, with ISPs exchanging the messages at the boundaries between their networks. Client programs under the control of a single client organisation may also be distributed in the network. This more general scenario is depicted in Figure 2.2

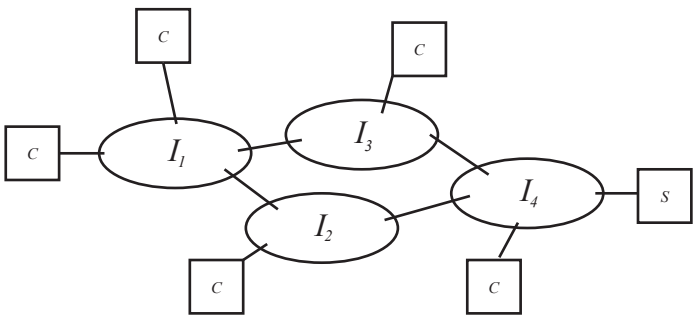


Figure 2.2: An ASP scenario with distributed clients and multiple network providers

Assuming that an application service already exists with the potential to meet some requirement of the client's, a service-provision scenario is established as follows:

1. the client first discovers the service, perhaps assisted by some directory technology such as UDDI [101];

2. the service will consist in part of electronic services offered at one or more particular points in some network(s). In order to access the service, the client must obtain access to some minimal subset of these service-provision points. Before beginning to use the application service, the client will assess the feasibility of obtaining such access;

3. the client assesses the feasibility of implementing or obtaining client software capable of using the application service;

4. assuming the client believes it feasible to access and use the application service, they may then wish or need to contact one or more parties offering access to the service. The client may need to obtain permission to use the service, as attempting to use a service without permission could be construed as malicious behaviour. The client may also wish to negotiate an SLA with the provider, as discussed further below. The service provider contacted need not necessarily be the actual provider of the service, but will take responsibility as such;

5. access to the service may be controlled by technical means, such as the need for a username and password. Assuming the client meets or undertakes to meet any necessary conditions, the service provider will arrange for any necessary credentials to be provided to the client;

6. the client will take whatever measures necessary to obtain access to the points of service provision;

7. the client will implement or obtain client software capable of using the application service;

8. the client will begin to attempt to use the service.

Application services may be offered for free, and require no contact between client and provider prior to an attempt by the client to use the service. Alternatively the provider may require the client to pay and/or enter into an agreement of some kind to govern the relationship.

The client will continue to attempt access the service until they choose or are forced to cease. This may occur in response to its permission to use the service lapsing, the service becoming unavailable for some technical reason, the client losing the capability to access the point of service provision, a deterioration in the client's relationship with the provider, or for any other reason.

Service-provision relationships vary in the amount of time that they last, from a single invocation, to years. In practice, the client and the provider may be the same party, using a service model as a convenient way to coordinate some larger activity. Alternatively, the client and provider might have had no prior contact whatsoever, and only interact via the network.

2.2 ASP risks

2.2.1 Risks to the client

A client is exposed to two major risks when employing an application service provided by a second party. First, I assume that a client is only ever motivated to use any kind of service because the service as advertised by the provider has the potential to deliver some value to the client. Therefore the client assumes the risk that the service will not meet some requirements necessary to deliver this value. The magnitude of this risk will depend on the reliance the client has on the service; at best the client may only have wasted its time, but the consequences may be far more severe. In any case the client will incur a cost, either directly or in terms of lost revenue. Such costs may occur occasionally or, if the service deteriorates but the client is unable to quit the service-provisioning relationship, over a long period.

Second, the client will generally have to make an initial investment to acquire or implement client software capable of using the service, or more generally to integrate the service into its IT infrastructure. If the service ceases to work altogether within the expected period of service-provisioning, degrades to the extent that it is no longer cost-effective for the client to rely on the service, or if for any reason the service provider prematurely withdraws permission for the client to access the service, then the client will have lost some opportunity to recuperate those costs.

These risks are illustrated in Figure 2.3. The graph depicts four flows of cash or value over time, related to a hypothetical service: the client's expected spend, the client's actual spend, the client's expected return and its actual return. The relationship between the client and the provider can be seen to be divided into two phases, the integration phase and the operation phase. During the integration phase, the client spends to integrate the service, and receives no value from the service. During the operation phase the client (potentially) incurs operating costs as a result of using the service, but has the opportunity to receive value in return.

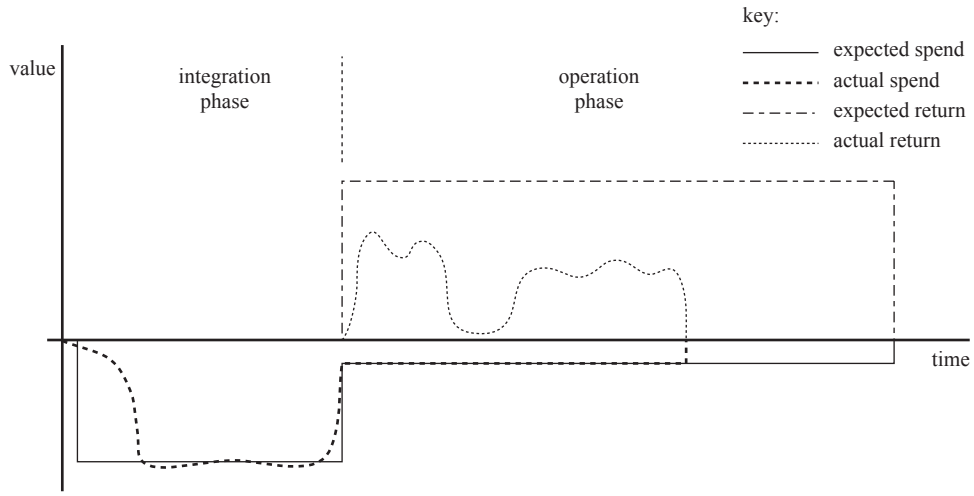


Figure 2.3: Example value flows in an ASP relationship

The graph depicts a relationship in which the operating period is shorter than expected for some reason, and the client is able to receive less value than expected during the operating period due to poor

service performance. Although the client has had to pay for the service for less time than they expected, the total amount they earn has been rendered unprofitable compared to its initial integration costs.

Both of these types of cost, which I refer to as *inefficiency* and *termination* costs, are opportunity costs. The client has lost an opportunity that they expected to have, as a result of using a service, to obtain some return.

An alternative way to view the costs incurred by the client over the lifetime of the service-provision relationship is depicted in Figure 2.4. Here the opportunity costs are represented as direct costs to the client. Note that because in this case the client expected to obtain a return from using the service at a constant rate, the cost due to poor service performance is the mirror image of the income shown in Figure 2.3. The termination cost is the lost income minus the saving in reduced operating costs. Since the income would not have been received all at once, this cost can be regarded as being spread over the interval following the actual termination of the service-provision relationship until the moment the client expected the relationship to end.

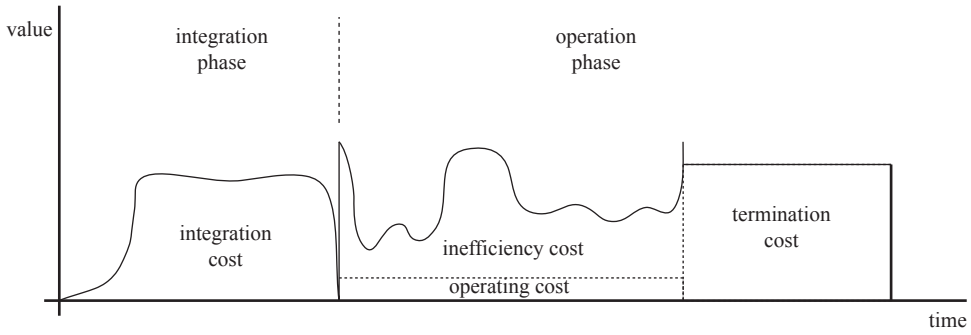


Figure 2.4: Possible costs to the client in an ASP relationship

This example makes plain the fact that using an outsourced service is a gamble. The client will make predictions concerning what they will spend on the service during the lifetime of the relationship, and concerning what they will be able to earn or receive in consequence. If these guesses are wrong, the client suffers.

The alternative to using an outsourced service is for the client to implement an equivalent service in-house. This will not always be possible, as the nature of a service may mean that not all parties will be capable of providing it. However, assuming that it is possible, it is helpful to consider the risks associated with this option in order to understand how outsourcing risks may be more or less problematic.

The risks associated with implementing a service in-house might plausibly result in a graph of exactly the same form as Figure 2.3, but the causes of spending and lost income will be different. Initial expenditure is now due to the cost of implementing the service rather than integration. The service might still generate less value than expected due to a lower than expected quality of implementation resulting in buggy behaviour, or an inability to correctly maintain the service. These will be inefficiency costs. If this becomes intolerable, the system may suffer premature obsolescence, essentially implying a termination cost. The similarity between the two scenarios makes intuitive sense. In both cases the client is relying on the competence of some party to provide a service: in the outsourcing case, that party is a second

party; in the in-house case, it is itself.

The key difference between these two scenarios is in the client's belief in its ability to predict the amount of risk involved in each. A party might think they have a better understanding of its own ability to implement and maintain a service than it does of the ability of a second party to deliver the service. It may therefore believe that it can better control opportunity costs due to poor service performance. Similarly, it might have more confidence in its own commitment to the service, and ability to regulate itself financially, than it does of a second party. Hence the party may assume that a service it implements itself will be available for as long as is required.

It is in the nature of ASP services that the parties tend to be distributed, with the main communication occurring over a network, most commonly the Internet. This tends to limit the amount of information that parties have about one another. For a potential client party, this makes it hard to assess the likelihood that a service will perform well, and be available as required over some reckoning period. With limited ability to quantify these probabilities, a prudent party will assume the worst. The overall risk of outsourcing will therefore be primarily related to the value of the service to the client, and the client will only be prepared to enter into low-value relationships, or will otherwise choose to implement services in-house. I argue that this is the main discouragement for parties wishing to make use of outsourced services, and the reason that service-oriented technology, such as middleware, has thus far found its principle application in structuring the activity within the administrative domain of individual large organisations, such as banks and retailers, or in very high-value relationships where costly risk-mitigation techniques such as due-diligence inspections or natural-language SLAs prepared by lawyers are feasible.

2.2.2 Termination risks

The graphs presented in the previous section allow a more complete consideration of the effect of the termination of a service provision relationship on a client party. The party will have at most three options: they may find and integrate a replacement outsourced service; they may implement an equivalent service in-house; or they may give up hope of receiving value due to the service.

Assuming the first service is performing adequately, then early termination by the provider will be a risk to the client, as it will lose the opportunity to recuperate its initial integration costs, and may incur the costs involved in integrating or implementing a replacement service.

However, if the original service is producing low returns, the client may wish to force early termination, either to cut its losses if the operating cost exceeds the value offered by the service, or because they prefer to invest in a new service what would otherwise have been spent in operating costs for the old. In this case being locked into a relationship with the first provider will represent a risk to the client, as the ongoing operating costs of the first service might be onerous or render the integration or implementation of a second service financially impractical.

2.2.3 Risks to service providers

A discussion of ASP risks would be incomplete without mentioning that permitting the client to access an application service may represent a risk to the other parties in the scenario, namely the service provider and the network-service provider, or ISP.

Perhaps the most basic risk that the client poses to these parties is that they will choose to use the service. If they do so they will inevitably consume network and computing resources, resulting in costs to the providers.

The providers will also have to invest money and effort in implementing the service and its supporting infrastructure (for example, the network). This will represent a cost to the providers, if they cannot find a way to profit from the service.

Unlike the risks to the client discussed above, these risks are actually quite easy for the providers to mitigate, and are therefore not limiting factors in the adoption of the ASP model. Since the service provider and network-service provider directly or indirectly control access to the service by the client, they can simply deny access to the client, preventing the client from using the resources. This can be used to hold the client to ransom, forcing it to pay for the privilege of using the service (what might be considered a 'pay-as-you-go' scheme), or obliging it to enter into an agreement that includes a commitment to reimbursement. In this latter case, the client will likely demand some reciprocal guarantees with respect to quality-of-service, and the commitment becomes an SLA.

2.2.4 The magnitudes of ASP risks

I make no assumptions concerning the magnitude of any of the risks described in this section. The magnitude of a risk is related to the likelihood of an event occurring and the degree of harm caused by that event: parties may or not behave reliably, particularly if there exists a financial incentive to cheat, so the probability of harm occurring is not bounded below; the costs of delivering a service, the gains to be made by using a service, and therefore the potential financial losses associated with service provision, are entirely dependent on the circumstances of the scenario, and are therefore not bounded above.

2.3 What is a Service-Level Agreement (SLA)?

A Service-Level Agreement (SLA) is an agreement between the client and the provider of some service. The term 'service-level agreement' implies that an SLA includes permission for a client to attempt to use a service. This is necessary as the client cannot expect to receive any level of service if they are not permitted to request service. It also implies that such an agreement will include at least some guarantee by the provider in relation to the service meeting certain requirements – some attempts to access the service by the client must result in some level of service, a refusal to provide service being no service at all. The nature of these requirements will depend on the type of the service, and on the outcome of negotiations between the parties.

However, simply defining what is required from a service by no means guarantees that that this will be provided. Therefore, an SLA primarily represents some guarantee to the client that the service will either meet the stated requirements or there will be consequences that will tend to compensate the client for the harm it suffers due to these requirements being missed.

If an SLA is protected by law, then it is a contract. However, not all SLAs are contracts, as SLAs are sometimes used to coordinate activities within large organisations. Such an organisation will fulfil multiple roles in the scenario, and the roles are therefore not filled by financially independent parties.

If the service fails to meet the client's requirements, the agreement may be broken. If the agreement

is a contract, the client may seek compensation in a court of law. If the agreement is more informal, the breach of the agreement may have other consequences for the relationship between the client and the provider, or the management of these parties.

In some cases, regarding the agreement as having been breached the first time that the service fails to meet some requirement is not practical. Instead, the provider may agree to provide some kind of compensation to the client in this event. Providing the compensation is paid, the agreement is not breached, and the parties are satisfied.

The association of consequences for the provider, potentially including penalties, with the violation of the client's requirements for the service implies that SLAs have the potential to mitigate risks to the client related to the behaviour of the service.

In such an SLA, the provision of compensation by the provider in the event of poor performance by the service becomes a constraint. This suggests that the SLA is not only concerned with the behaviour of the service, but also that of the service provider.

SLAs may also function as a means for the provider to charge the client for using a service. This helps to mitigate the financial risk to the provider inherent in developing the service originally. An agreed charging scheme will also become necessary, because by entering into an SLA a provider will typically agree to suffer negative consequences as a result of withholding access to the service from the client. This may effectively eliminate this as a mechanism available to the provider to mitigate the risk that the client will choose to use the service, hence implying costs to the provider. The provider of a service may therefore, in some service-provision relationships, reasonably seek to impose conditions on the client, for example that the client pay to use the service, to mitigate risks of this kind to the provider.

The client may also have the potential to behave in a manner more or less objectionable to the provider. The provider may agree to tolerate some bad behaviour in return for some kind of compensation, or a relaxation of their own obligations.

An SLA may therefore be a mechanism by which either party may become liable to provide compensation to the other party. In this respect an SLA can represent an additional risk to either party, which is that they will by some means be forced into a situation where it is unavoidable that they must incur a penalty according to the terms of the SLA. Clearly for SLAs to be an attractive means to mitigate risk, they must be as non-exploitable as possible.

Either party may wish to establish the right to terminate the agreement under certain conditions. If payment is required, the client will wish to quit the agreement if service performance is consistently bad. The provider may wish to withdraw permission for the client to access the service if the client behaves consistently badly.

Similarly, either party may wish to receive guarantees as to the lifetime of the agreement. The client may be benefiting from the service, and wish this to continue. The provider may wish to safeguard payments for the service to cover an initial investment in the service or turn a profit. Penalties for either party may be related to the early termination of the agreement by that party.

To summarise, an SLA provides permission for the client to access a service in some manner that

is acceptable to both client and provider, and will also define conditions relating to the behaviour of the service, with the provider considered to be responsible for violations of these conditions. Additionally, conditions may be placed on the behaviour of the client and the provider. Violation of a condition may result either in a breach of the agreement, in which case the consequences for the parties in their continuing relationship will no longer be explicitly governed by the SLA (although the SLA and the nature of the breach may be highly pertinent to subsequent events in the relationship between the parties), in an obligation for the responsible party to perform some compensating action, or in a modification of the effect of other conditions in the SLA.

Henceforth I only consider SLAs with a concrete representation, not word-of-mouth agreements, or de-facto agreements. When I refer to an SLA below, I am referring to a concrete representation of the agreement.

SLAs represent an agreement between two parties. The conditions encoded in an SLA are therefore not the whim of any one party, but a result of negotiation between the parties. It is nevertheless possible for a service provider to offer standardised commodity SLAs for its services, into which a client may choose to enter.

2.4 SLAs for application services

In the preceding sections I described the risks to parties in the ASP scenario, and suggested that the risk to clients significantly inhibits the adoption of the ASP model; I also described the potential of SLAs to mitigate risks to parties in a service provisioning relationship. I now briefly discuss the particular role of SLAs for ASP.

SLAs clearly have the potential to be used to mitigate the risks in the ASP scenario, by associating compensation for the client with poor service performance, compensation for either party with early termination of the service, and by providing a mechanism for the provider to charge for the service.

Figure 2.5 reprises the example service-provision relationship described in Section 2.2. Now an additional cash-flow is depicted representing compensation payments paid by the provider to the client according to the terms of an SLA. Note that compensation is paid in response to poor performance of the service, and in the event of early termination of the relationship, and goes some way to balancing the inefficiency and termination cost incurred by the client. The client's operating costs for the service may now be (at least partially) explained in terms of payments required by the provider under the terms of the SLA.

It cannot be assumed that SLAs will be able to mitigate all risk to the client in all circumstances. The value of a service to a client will vary, and the service provider should not necessarily be expected to indemnify its clients against all kinds of risk. Moreover, in situations in which the service provided is hard to reproduce, or in which the existence of the service offers a business opportunity to the client, the service provider may not have to offer strong guarantees in order to retain its clients. However, even in these circumstances, an SLA can be used to adjust the level of risk that each party assumes.

On the other hand, the use of SLAs in the ASP scenario has the potential to make outsourcing significantly preferable to implementing services in-house. By associating penalties with poor service

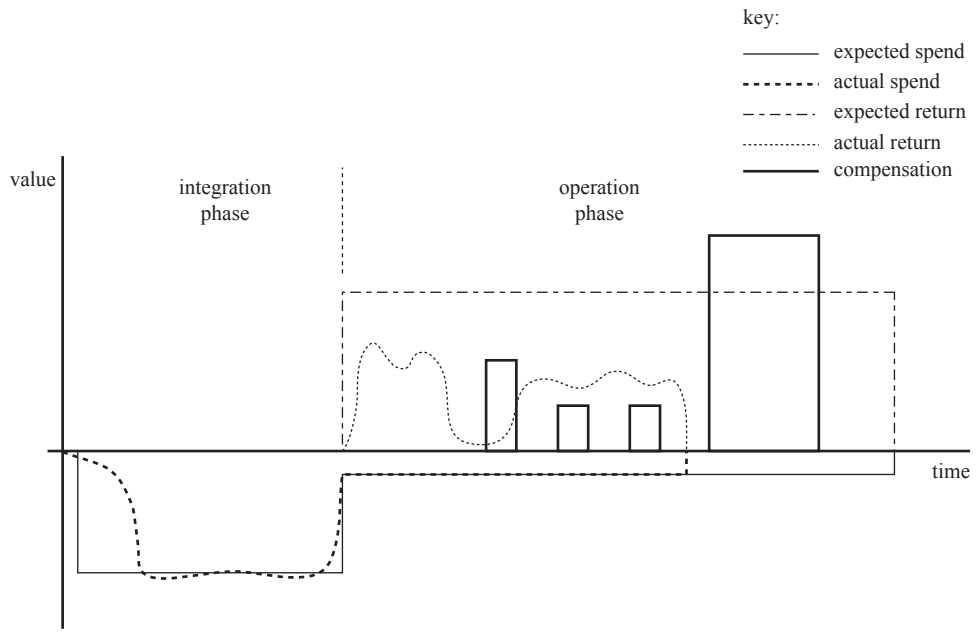


Figure 2.5: Flows of value in an ASP relationship with compensation payments governed by an SLA

performance, or early termination of the service-provision relationship, an SLA provides only minimal additional information to the client concerning the probability of these things occurring; the SLA only indicates that the provider expects that the conditions will be met or is prepared to take the consequences otherwise. However, the penalties associated with SLA conditions can mitigate the risks, and have the potential to do so totally. In contrast developing a new service will always imply some risk.

Indeed, total mitigation of risk may not be necessary to make outsourcing more desirable than implementing services in-house. Implementation will tend to be more expensive than integration, so outsourcing may be preferable providing the magnitude of the financial risk is favourable in comparison to the difference in start-up and operating costs.

As discussed in [97], SLAs are used in current industrial practice to manage relationships between application-service providers and their clients. However, as described in Section 2.10, the types of conditions included in practice do not necessarily systematically address the risks to the participants. Poor SLAs may be better than no SLAs at all. Also, the importance of having an SLA is diminished if risk is mitigated by other means. In high-value service provision relationships the parties may be much more prepared to invest in legal services to assist in the management of the relationship. Consequently either good quality, legally-binding SLAs will be produced at great expense, or litigation or arbitration can be relied upon to settle disputes satisfactorily.

However, it seems clear that lower-value service-provision relationships could also benefit from the use of SLAs. This may allow the ASP model to be used in industry where previously it was infeasible, due to the level of risk implied by the situation and the relatively high cost and poor quality of industrial standard SLAs. Moreover, the cost of using SLAs even in high-value relationships could be decreased if a repeatable way to produce good SLAs can be found.

A logical approach to addressing this problem is to provide language support for authoring SLAs for

ASP services. A substantial amount of previous research work has focussed on this approach, discussed in detail in Chapter 8. The contribution of this thesis is to demonstrate that improvements on this previous work are possible by focussing on the role of SLAs in the ASP scenario in mitigating risks.

Providing good language support for any purpose depends on anticipating what needs to be expressed. I next examine the conditions that could be reasonably required over application services and the client, based on the assumptions that I have thus far made about ASP scenarios.

2.5 Conditions relating to application services

In Section 2.1 I presented an abstract model of application services. Although in principle the client of a service may find any behaviour either favourable or unfavourable, by considering the scenario it is possible to draw some conclusions as to likely conditions the client will wish to place on these types of service. By then considering the risks that a provider would expose itself to by agreeing to these conditions, it is also possible to anticipate what conditions the provider will in turn require from the client.

A client of a service should reasonably only be concerned with the behaviour of a service in so far as it affects the client. The internal behaviour of the service should be the responsibility and concern of the service provider alone, provided that the results delivered to the client are satisfactory. In this respect appropriately written SLAs may be a more attractive risk-mitigation strategy than say due-diligence inspections, as the client need not purchase expertise in the business practices or technology used by the provider.

Referring back to the scenario depicted in Figure 2.1, it is clear that two kinds of behaviour of the service may affect the client. First, the client may receive information from electronic services via the network, either in the form of responses, or information pushed to electronic services implemented by the client software; and second, the service may take other actions, apart from those related to electronic services, whose consequences eventually affect the client.

For example, consider the purchase of a book from the online bookshop `Amazon.co.uk`. A purchaser will interact with the service via its website, browsing its stock and in due course submitting an order. This interaction will consist of a sequence of webpage responses and requests and will be transacted entirely through the medium of the network. If the submission of book purchase requests were a matter of urgency for the client, then an SLA could be established to constrain the timeliness and reliability of responses to page requests. The interaction will also result in activity on Amazon's part to fulfil the order. A book will be retrieved from a warehouse, or ordered from a third party, and will be packaged and dispatched via a postal service. This latter type of behaviour does not require interaction with the client over a network, but still ultimately affects the client when the book is delivered, or it fails to arrive when expected.

To distinguish this kind of behaviour from that related to electronic services, I henceforth consistently refer to it as *real-world behaviour*, although I recognise that electronic services also exist in the real world.

Communications originating from a service have two main attributes that the client could seek to

constrain. What is returned, and when it arrives. Conditions related to the interval between a service request and the time of arrival of a correlated response are variously referred to as performance, latency or *timeliness* conditions. These may also apply to pushed information, if the information is provided as an asynchronous consequence of an earlier request. Alternatively, pushed information may have to conform to some schedule.

Because the client has no access to the implementation of the service, its expectations concerning the behaviour of the service will depend on a description of the service given to them by, or negotiated with, the service provider. Before entering into an SLA the client will make the choice to integrate the service into its own operations on the basis of this description. If the service subsequently behaves in a manner other than that described, the client is likely to suffer. Hence, a condition that the client will want to protect in an SLA is that the service either behaves as described to a high degree or client will be entitled to receive compensation. Such conditions are normally called *reliability* conditions.

Communications via electronic services have no other attributes, so I conclude that the client will be primarily concerned with timeliness and reliability conditions relating to these services, and with conditions relating to the real-world behaviour of the application service as a whole.

Conditions concerning the timeliness and reliability of a service may be highly diverse. For example, a client may wish to require that failures that occur at a particular crucial point in a business process are very highly penalised, or similarly, that delays at peak times incur heavy penalties. I make no assumptions concerning the nature of these requirements. I also assume that the client may have arbitrary requirements concerning the real-world effects of a service.

As mentioned above, by entering into a service-provisioning relationship a service provider exposes themselves to the risk that the client will choose to use the service, implying a cost to the provider related to the resources required to deliver the service to the client. In the absence of any SLA related to the service, this risk may be mitigated by withholding the service. However, reliability and timeliness conditions applied to a service-provider in an ASP SLA reduce the effectiveness of this as a risk management mechanism for the provider. It will therefore be necessary for the provider to implement a charging scheme in an ASP SLA, and for reliability and timeliness conditions applied to the provider to be conditional on the client meeting their obligations under this scheme.

If reliability and timeliness conditions are applied to the provider of an electronic service, then the provider assumes an additional risk due to the finite capacity of such services. Characteristically, the timeliness of an electronic-service will decrease drastically once some critical resource required to service requests, such as a processor or database, approaches 100% utilisation [63]. At this point the length of queues of requests awaiting access to the highly contended resource begin to increase dramatically, with waiting times increasing proportionally. Also, due to the necessarily finite capacity of queues in the implementations of electronic-services, if the volume of incoming requests remain high it will eventually become necessary to begin ignoring requests, as no further queue capacity will be available. This behaviour will manifest itself as unreliability in the service.

Little's law dictates that the mean length of a queue for a system in equilibrium (in which the mean

request rate is lower than the mean service rate) is the product of the request rate and the mean response time (the reciprocal of the mean service rate) [40]. Since in an electronic service, the provider is unlikely to be able to improve the response time of the critical resource at runtime, the only way for to control the queue length, and hence the overall time spent in the queue, is to limit the rate of requests. However, the rate of requests is controlled by the client. Therefore it is possible for a client to attempt to exploit an ASP SLA by increasing the rate of requests.

This risk to the provider can be mitigated in an SLA by applying a condition to the client that requires a limit on the rate of service requests. The consequences of violating such a condition may be various, including: requiring the client to pay a penalty to the provider; rendering the client ineligible to receive compensation for violations of timeliness and reliability conditions in the SLA; or breaching the SLA altogether. I refer to such a condition as an *throughput* condition.

Conditions of any kind may relate instances of some kind of bad behaviour to an obligation for a party to pay a penalty, or otherwise perform some compensating action. However, for the party in question to become aware that a violation has occurred they must periodically check whether the conditions in an SLA have been violated, or be informed of a violation by another party that has performed this checking. I refer to this process as *administering* the SLA.

It will often be necessary for an SLA to explicitly state the obligations of the parties with respect to administering the SLA. A condition must describe the compensation associated with a violation, and may also place a constraint on when this compensation should be delivered, or else the liable party could defer the provision of compensation indefinitely without violating the agreement. Clearly, such a constraint would implicitly require the party liable to deliver compensation to administer the SLA at some point between the violation occurring and the compensation becoming due.

However, parties may not wish to continuously administer the SLA, so it may be preferable to define deadlines for compensation in relation to scheduled administrations, or administrations triggered by specific events. Even if obligations to deliver compensation are directly triggered by violations (rather than by the administration of the SLA), the parties may make genuine mistakes in calculating their own liability to pay penalties. This should not necessarily result in the immediate breach of the SLA, so in this case it is convenient to regard administering the agreement as a consensual process involving a component of negotiation, and the SLA will have to contain details of how this is to be achieved. One possibility is to include provisions for the parties to negotiate a reconciled account of service provision from which violations will be calculated. The relationship of administration conditions to the monitorability of an SLA is discussed further in Chapter 5.

2.6 Systems of SLAs for ASP

Referring once more to the scenario presented in Section 2.1, I observe that electronic services may be delivered to the client over one or more networks controlled by network-service providers. These providers may be independent of the application-service provider, but it is clear that the behaviour of the networks has the capability to introduce delays and faults into the communications between the client and the electronic services constituting the application service. This is precisely the risk that the client is

seeking to mitigate through the use of SLAs.

However, since more than one party may be responsible for degradation of the quality of the service, as received by the client, with what party should the client enter into an SLA? Also, there are obviously two very different types of service being provided in the scenario. The application-service provider provides an application service, and the network-service providers provides the service of moving information around their networks. In the previous section I described the kinds of conditions that the client will wish to associate with compensation using SLAs, but might not more conditions be needed to constrain the behaviour of the network? If a fault occurs, how will the client or any other party know who is responsible for it, and hence who should pay compensation? As discussed in Section 2.1, access to electronic services will only be offered at one or more defined points in some network or networks, commonly the interface of the computer providing an electronic service to the network in which it resides. The client may need to enter into additional agreements simply to obtain permission to access this point in the network.

I address these questions in Chapter 5, using the requirement that SLAs be monitorable (introduced below) as a way to identify good choices of SLAs for the scenario. Here I merely note that any given ASP scenario may require not merely one SLA, but a system of SLAs, in order to mitigate the risks identified for the parties without introducing unacceptable new risks. The SLAs in a system will contain conditions that, in combination, will act to deliver compensation to the injured party when a harmful event occurs.

2.7 Requirements for systems of ASP SLAs

In this section I consider requirements for systems of SLAs capable of mitigating the risks identified in Section 2.5. I then consider the requirements that languages for expressing such SLAs should meet, and also requirements for the specifications of such languages, the quality of which have a significant impact on the usefulness of the languages they define. The purpose of elaborating these requirements is to clarify what is meant in the thesis statement by ‘practical language support’ for ASP SLAs. Such support is clearly more practical if it is oriented toward writing useful ASP SLAs, so the requirements for systems of SLAs in the scenario must first be understood. By explicitly elaborating the requirements, I also provide a basis for the comparison of ASP SLA languages and motivate the design of the SLAng language described in later chapters.

The requirements in this and subsequent sections are expressed as absolutes, as would be met by an ideal system of SLAs, an ideal language and language specification. However, for each requirement varying degrees of satisfaction are possible, and incomplete satisfaction of a requirement does not render an SLA, language or language specification completely useless. SLAs are a measure for controlling the level of risk assumed by the parties involved in an outsourcing scenario, and the use of even imperfect SLAs may mitigate this risk to some extent. Also, trade-offs between requirements may be necessary. For example, a highly expressive language may be hard to use.

As discussed above, multiple SLAs may be required to insure the service experienced by the client at the location in the network from which the client wishes to access the service. In this section I therefore

consider the requirements for systems of SLAs as a whole.

2.7.1 Conditions appropriate to electronic services

As discussed above, the principal role of SLAs in the ASP scenario is to entitle the client to receive compensation when its requirements with respect to the behaviour of the service are violated, or to provide the client with justification for terminating an SLA without penalty. Due to the nature of electronic services, these requirements are likely to include reliability and timeliness constraints, as well as constraints on the real-world behaviour of the service. This is captured by the following requirements:

SLA 1 (Service Conditions) *The system of SLAs should entitle the client to either receive compensation, vary some SLA or SLAs in an agreed manner, or provide them with the opportunity to quit the system of SLAs without penalty, when the behaviour of the service, in so far as this effects the client, violates some anticipated requirement of the client, potentially including timeliness and reliability requirements.*

The SLAs should address the risks to the providers implied by offering these guarantees, and therefore being obliged to interact with the client. This includes the risk that the client will attempt to overwhelm the service with requests.

SLA 2 (Client conditions) *The system of SLAs should entitle any service providers involved to either receive compensation, vary some SLA or SLAs in an agreed manner, or provide them with the opportunity to quit the system of SLAs without penalty, when the behaviour of the client, in so far as it effects the service, violates some anticipated requirement of the provider, potentially including request-throughput limitations.*

The system of SLAs should also allow the service provider and any network service providers to receive compensation for providing their services.

SLA 3 (Charging) *The system of SLAs should make the service provider and network-service provider liable to receive compensation, in return for their contributions to providing the service to the client at the client's preferred point of service delivery, if the providers require compensation.*

Note that the system of SLAs cannot guarantee that any party will receive compensation when entitled to it, as this is dependant on the ability of the liable party to deliver compensation, which is by no means guaranteed.

The SLAs should address termination risks to the parties.

SLA 4 (Termination) *The system of SLAs should make any party liable to receive compensation when one or more SLAs in which they participate are terminated prematurely by another party.*

2.7.2 Protectability

In an ASP scenario governed by SLAs, if any party is entitled to receive compensation, then one or more parties with whom they have an SLA will be liable to pay. The capacity of a system of SLAs to establish these rights and liabilities is therefore likely to be a point of contention between the parties in the event that they are asserted. I have already assumed that parties to SLAs cannot be relied upon to act honestly,

particularly if they have a financial incentive to do otherwise. The effectiveness of the system of SLAs as a mechanism for controlling a party's exposure to risk, is diminished if in the event of such contention, disagreements cannot be resolved according to the original intent of the agreement.

I refer to the ability of an SLA to entitle parties to receive the pre-agreed compensations under the pre-agreed circumstances as the *protectability* of the SLA. This is because for the SLA to come into force, the parties to it must have agreed to its provisions, and any deviation from those provisions represents a violation of that agreement.

SLA 5 (Protectability) *All SLAs in a system of SLAs must be protectable.*

Resolving a disagreement concerning the intent of an SLA with respect to a given situation relies on: recovering that intent from the concrete representation of the SLA; obtaining evidence concerning the behaviour of the service and parties relevant to the determination of compliance with the SLA; convincing all parties to the agreement, or any arbitrator of the agreement, of the validity of the evidence; and determining whether the evidence represents compliance with the SLA, or what future action is required to ensure compliance.

Protectability hence implies two categories of derived requirements: SLAs should be precise and understandable so that their intent can be recovered and interpreted in relation to evidence; and they must be monitorable, so that it is possible to obtain pertinent, reliable and convincing evidence.

2.7.3 Precision

An SLA must express the true agreement with respect to service levels between the parties to the agreement, and it must be possible to understand the SLA at any time after it has been written:

SLA 6 (Understandability) *SLAs must be understandable, so that all parties can verify that an SLA correctly captures their intent with respect to the agreement, and so the intended effect of the agreement can be easily retrieved in the event of a disagreement related to the award of penalties.*

SLA 7 (Precision) *SLAs must be precise, so that their intended effect is unambiguous in the case of any disagreement related to the award of penalties.*

2.7.4 Monitorability

I refer to the gathering of evidence to determine if an agreement is being violated as *monitoring* the agreement. The degree to which a system of SLAs facilitates or hinders monitoring is the *monitorability* of the system. Informally, monitorability may be affected by the choice of what events are pertinent to an SLA, as some events are intrinsically easier for certain parties to monitor than others. Clearly, the more monitorable a system of SLAs, the easier it is to protect the SLAs in that system.

SLA 8 (Monitorability) *The system of SLA should be as monitorable as possible.*

Chapter 5 introduces a formal theory of monitorability. I show that monitorability affects the choice of SLAs in the scenario, and hence their design, and the language support required for them.

Another aspect of monitorability is the effect of error and uncertainty on measured quantities. Measurement of any quantity in the physical universe is subject to a degree of error, which manifests itself

as a lack of confidence in the value obtained due to the possibility of problems occurring during the measurement process, and frequently also as a difference between the measured value and the true value of the quantity being measured. A measurement may also contain a degree of uncertainty due to the precision with which it is stated.

The intent of an SLA is to place constraints upon the true behaviour of a service, the violation of which will have consequences. However, assessing the violation of these constraints will require measurement of the service, and the calculation of violations based on these measurements. This raises two problems related to monitorability. First, an appropriate basis for the calculation of violations must be established. If conditions are stated in relation to the true behaviour of the service, then they must be formulated to accommodate a degree of error in their calculation, because measured values, not true values will be used to assess them. Alternatively, if conditions are defined in relation to measured values, then a condition must ensure that the measured values are tolerably close to the real value of the quantity measured, otherwise parties may choose to purposefully vary the error term in reported measurement values in order to misrepresent the behaviour of the service.

SLA 9 (Error) *SLAs should accommodate measurement error and uncertainty, either by only setting conditions on measured or agreed quantities, with a description being given of how the measurements are to be taken or the agreement reached, or by specifying acceptable degrees of confidence and margins for error on constraints over actual physical quantities.*

The second problem is that quantities of error and uncertainty present in measurements may accumulate when calculations are performed on measurements. For example, the error term of the sum of a set of independent measurements is the sum of the error terms of the measurements. If conditions are stated with respect to the real behaviour of the service, with a requirement for a minimum degree of confidence associated with the calculation of violations, it will be necessary for the parties to the SLA to determine how the error in their measurements accumulates to give a resulting error in their determination of the violation. Depending on the formulation of the condition, this may be extremely difficult. This is one example of a broader requirement for SLAs: given all pertinent evidence, it should be feasible to determine whether a condition has been violated.

SLA 10 (Feasibility) *SLAs should only include conditions for which violations can feasibly be calculated, given all pertinent evidence.*

2.7.5 Cost

The use of SLAs in an outsourcing scenario implies additional costs for both client and provider. These should be minimised.

SLA 11 (Cost) *SLAs should be as cheap to produce, protect and administer as possible.*

Other costs related to the consequences of having an SLA may be incurred by the parties. Requirements related to these are discussed in the next section.

2.7.6 Machine readability

Using an SLA in an electronic service scenario will tend to introduce requirements for monitoring, service adaptation and negotiation. An obvious approach to reducing the cost of using an SLA is to use the parameters of the SLA as inputs to mechanisms for automating tasks of these types.

If any degree of automation is to be applied to attempt to meet the terms of an SLA, or to manage or negotiate SLAs, then some machine-readable form for all or part of the SLA will be desirable.

SLA 12 (Machine readability) *SLAs should be expressible using an intrinsically machine-readable syntax. This requirement should not compromise understandability.*

A machine readable representation of an SLA may not be appropriate for human comprehension and vice versa. This raises the prospect of several representations of the same SLA existing. Under these circumstances, it should be clear what document represents the agreement for the purpose of determining violations. This requirement is related to the requirement that SLAs be precise.

SLA 13 (One definitive form of agreement) *If multiple forms of an SLA exist, they should be provably equivalent, or it should be clear which is the definitive form.*

2.7.7 Non-exploitability

SLAs may associate violations of behavioural constraints with penalties for the party responsible for the violation. However, it may be possible for one party to behave in such a way as to force another party to commit a violation and pay a penalty. If an SLA is obviously exploitable, there will be a disincentive for some party to agree to it, eliminating its usefulness as a mechanism for mitigating risk in the ASP scenario. If the SLA is exploitable, but not obviously so, then it may be hard to understand, or not adequately analysable, in violation of other requirements stated here.

SLA 14 (Non-exploitability) *SLAs should be not be exploitable.*

It may be that the obligations or constraints expressed in an SLA imply other obligations or constraints that are not explicitly stated in the SLA. This could be because these implications are only of concern to a subset of the parties to the agreement, or because stating all of the implications of the agreement would be inconvenient to the expression of the agreement. However, it should still be possible for each party to become aware of those implications of concern to them.

Moreover, the use of SLAs complicates development and maintenance of services for service providers. SLA information may therefore be used during development and planning. These scenarios suggest that SLAs should be amenable to analysis.

SLA 15 (Analysability) *SLAs should be amenable to analysis to reveal implications that are not explicitly stated.*

2.8 Requirements for ASP SLA languages

Since we are only considering SLAs with a concrete representation, these SLAs must be written in some language, either a natural language, a technical language or some combination of both. In this section

we consider requirements on languages or combinations of languages capable of expressing all SLAs required in a system of SLAs meeting the requirements described in the previous section.

Language 1 (Expressiveness) *The language must be capable of expressing all SLAs in a system of SLAs meeting the requirements specified in Section 2.7.*

To understand an SLA written in a technical language, it is necessary to understand the semantics of the language. An SLA language hence assumes some of the burden of expressing the intent of an SLA. Therefore all requirements related to the precision of an SLA are also requirements of SLA languages:

Language 2 (Understandability) *To understand an SLA written in an SLA language it is necessary to understand the language. The language should be structured so that it is easy to understand.*

Language 3 (Precision) *The meaning of an SLA is dependent on the semantics of the language in which it is expressed. Therefore, if the SLA is to be precise in its meaning, then the semantics of the language must also be precisely defined.*

The cost of producing an SLA in a technical language is related to the features of the language. A number of requirements for the language may be derived from the requirement that SLAs be as cheap to produce as possible.

SLA languages should support the process of creating SLAs in similar manner to that in which programming languages support the creation of programs. The syntax should restrict the set of SLAs that can be expressed to eliminate some illogical or ill-formed SLAs. The semantics should support the creation of consistency checks to detect SLAs that are illogical.

Language 4 (Restrictiveness) *The language should exclude SLAs that do not meet the requirements specified in Section 2.7.*

The language should be easy to write:

Language 5 (Ease of use) *In addition to being easy to understand, the syntax should be easy to write, possibly with the aid of tools.*

The language will ideally be used to write multiple SLAs. Those SLAs will have some features in common, and some features that vary. The need to repeatedly rewrite common SLA terms would be burden on an SLA writer, so it is preferable to encode common SLA features into concise language constructs:

Language 6 (Power) *Because the SLA language is only defined once, but may be reused in multiple SLAs, as much of the burden of expressing the SLA as possible should be placed on the SLA language, except where this is incompatible with requirements for understandability for either the SLA or the language.*

In Chapter 3 I develop a metric for domain-specific languages to formalise this informal notion of language power, and assist in the comparison of SLA languages with respect to this requirement.

In relation to the requirement that SLAs should be machine readable:

Language 7 (Automatability) *It should be possible to produce tools that take SLAs expressed in the language as their input. The tools should rely for their functionality only on the specification of the language, so that anybody who has access to the language definition can reuse the tools successfully.*

The design of an SLA language may support the analysis of SLAs expressed in the language.

Language 8 (Analysability) *The semantics of the language should be oriented towards that of known analysis models, provided this is compatible with expressing the true requirements of the client, and any additional constraints required to avoid exploitability.*

2.9 Requirements for ASP SLA language specifications

Clearly any SLA language should be defined explicitly because that definition will need to be delivered to the users of the language, and referred to when determining the intent of an SLA. The artefact defining the language is the language specification. I refer to the language in which the SLA language specification is written as the meta-language.

The quality of the specification will effect the usability of the SLA language. In this section I therefore consider requirements for specifications.

In many cases the language specification will be the sole source of information available to a user of the language, so the specification should capture all information of relevance concerning the language.

Specification 1 (Completeness) *The specification should fully define an SLA language meeting all of the requirements specified in 2.8.*

Because SLAs must be interpreted with respect to the language specification, the specification also inherits the precision requirements applying to SLAs.

Specification 2 (Understandability) *The specification must define the SLA language in a way that is understandable.*

Specification 3 (Precision) *The specification must define the SLA language in a way that is precise.*

The SLA language definition serves as the reference for any party implementing tools to manipulate SLAs defined in the SLA language. The language definition may therefore be an artefact in some software development effort.

Specification 4 (Automatability) *The meta-language employed in the specification should be defined in such a way to assist the development of tools that rely on the SLA language definition, for example, by offering a formal definition of the SLA language that could be used as the input to software engineering tools.*

2.10 Other views on requirements for SLAs

SLAs are currently employed in a variety of contexts, including ASP, although the practice is far from ubiquitous. Also, SLAs are typically not strong, legalistic agreements, as I have assumed in this chapter, but are instead support a broader Service-Level Management (SLM) approach. In this section I review the use of SLAs for SLM, and also prior academic work that has discussed SLAs for ASP.

SLAs for IT services are currently most usually a component of an SLM management approach [126]. The term *IT services* in this context encompasses to a much wider range of services than the application-services that form the foundation for our work on SLAs. IT services include the provision of any type of technical support for a business, including the maintenance of hardware, network and software environments, technical support and also the provision of application services, and are also referred to as Operation Support Solutions (OSS) [51]. SLM is primarily concerned with maintaining the relationship between the business and the IT service provider, largely through the use of SLAs. Providers are either IT departments within an organisation, or companies specialising in the provision of IT services.

In [126] the benefits of SLM are stated as being the following:

- client satisfaction – SLAs force a client to state their requirements;
- managing expectations – SLAs document client requirements, preventing 'expectation creep';
- resource regulation – IT service providers can control the demands of their clients using SLAs;
- internal marketing of IT services – A history of meeting SLA conditions can be used as a marketing device, improving the reputation of a service provider;
- cost control – Without knowledge of true client expectations IT services may tend to be over provisioned;
- defensive strategy – IT service providers meeting SLA conditions can avoid unwarranted criticism from users.

SLM generally assumes long-lived and high value relationships between client and provider. These assumptions modify the requirements for SLAs considerably. In SLM, SLAs are created following a process of feasibility analysis and negotiation, that in itself takes time and is costly. A typical term for an SLA is cited as being two years, because any shorter and the cost of producing the SLA would be prohibitive. Writing the SLA precisely is less important than negotiating a realistic agreement between the parties, so the need for a technical language is not emphasised. SLAs for SLM also tend to make availability an objective of prime concern. The emphasis on this comparatively gross and unmonitorable property (as discussed in Chapter 5) indicates that SLAs in SLM are used to guarantee tolerable levels of service, rather than the satisfaction of rigourous constraints.

The SLM approach is philosophically different from my own in that it relies on strong assumptions about the culture in which an SLA is to be deployed. In contrast, my own assumptions, that the duration of an SLA may be short, its value low, and the requirements of the client arbitrarily specific and rigourous (with the agreement of the provider), reflect a more technical approach intended to offer a risk management approach to the client in a wider range of circumstances, particularly when the parties may not be cooperative and trustworthy. I have also developed requirements that emphasise the importance of SLAs as technical artifacts supporting service development, deployment, composition and analysis, rather than simply management documents. However, although more technical than the SLAs required or used in

current practice for SLM, SLAs meeting my requirements can potentially fulfil the same role with several advantages. The use of a formal language to specify SLAs can be expected to assist in reducing the costs associated with SLM when attempting to manage outsourced services, reduce negotiation time by identifying key performance indicators and reduce the cost of SLA preparation.

The following types of condition for electronic-service SLAs were proposed in an review of industry SLAs provided by the industrial partner to the European project TAPAS [97]:

1. Timeliness – the amount of time the service takes to complete should be constrained.
2. Throughput – the client should not be able to overwhelm the service with requests.
3. Availability of the service – expressed as a proportion of the period of the agreement.
4. Maintenance and service schedule – when the service should not be accessed, or may legitimately underperform.
5. Backup of data stored by the service
   (a) Solution – the particular software product employed to backup the date.
   (b) Frequency – when the backups should be performed.
   (c) Facilities – where the backups should be stored to ensure their safety.
   (d) Access – how the client can get at data backed up on its behalf.
   (e) Data types to be backed up
6. Security policy
7. Monitoring and reporting policy – how conformance to or violation of the SLA should be reported.
8. Failure clauses – what should be done when the SLA is violated.

This list includes a number of conditions relating to the service in addition to timeliness and throughput, and reliability is not explicitly mentioned. However, the list provides some validation of this otherwise theoretical discussion of the ASP scenario. Availability and data backup conditions can be seen to be attempts to constrain the overall reliability of the service, as the service is not delivering its advertised results if it is inaccessible or if the data that it operates upon is corrupt. Security policy may also define an aspect of the functional behaviour of the system.

A maintenance and service schedule represents a variation of the timeliness, throughput and availability conditions. The inclusion of failure clauses in the list indicates an understanding that SLAs are only useful if they have some consequences, which I have assumed in this work relate to compensating injured parties for harm received due to violations of SLA conditions.

However, the list is also somewhat naïve. No mention is made of otherwise constraining the functional behaviour of the service. Also, compliance with conditions such as specifying the frequency of backups is hard to check because these activities are internal to the service and therefore cannot be monitored by the client.

Security risks, for example related to non-dissemination of confidential information, may be a disincentive for a party to use an outsourced application service. SLAs may potentially offer some assurances to a client that the provider will either prevent security violations or pay compensation. However, I have elected to exclude the consideration of risks of this type from the scope of this work. I discuss the requirement for further research into this matter in Section 9.3.4.

In Appendix A I provide a review of previous languages with the potential to express SLAs for ASP, summarised in Chapter 8. It is notable that in most of this work requirements are not discussed in detail, suggesting that they may not have been considered in detail. However, there are exceptions.

Requirements for contract languages are considered in relation to the Business Contract Language (BCL) in [74]. Here the authors touch on high-level requirements for business contracts such as the inclusion of security provisions, access-control and obligation policies, specifications of standards for precision of measurements, feasibility of checking contract provisions, conditions relating to administration, and the need to have flexible specifications for states, events and temporal constraints. In the statement of some of these requirements there seems to be a confusion between defining what is needed in a contract and how it should best be expressed. For example, the authors state that a contract language should be able to describe behavioural patterns with a similar expressive power to that normally associated with process algebra. In fact, this is unlikely to be expressive enough for all cases. However, the authors generally advocate a high level of expressiveness for the language, which accords with my own observation that conditions may vary according to boundlessly variable factors. The inclusion of requirements related to policies (e.g. access control), also indicates a wider intended scope than merely SLAs. Like my own work, the authors observe that trust between parties may not be absolute. However, no consideration is given as to how reduced levels of trust may interact with monitoring, or policy rules. Instead it is used to motivate requirements for security provisions. Overall, the requirements stated largely accord with my own, although they are not systematically enumerated, and due to a lack of a definitive specification for BCL, it is hard to assess how many of these requirements have been met by the language. Certainly, no mention is made of measurement error in subsequent papers on the topic.

Requirements for agreements of several types are also considered in work related to the X-contracts language [69]. This work focusses on the use of agreements at runtime, and therefore supports my observation that SLAs may themselves be useful software-engineering artifacts. However, it also lists requirements for agreements throughout their lifecycle, which is divided into five phases: specification; provision; monitoring; adaptation; and resolution. Specification requirements include the need for a balance between expressiveness and simplicity and provision for the definition of penalties. Monitoring requirements include scalability, which may be considered to be related to my own requirements for feasibility, and techniques to enhance trust. Resolution requirements include the need for termination or renegotiation of SLAs, customer-credit schemes and non-repudiable exchange of information. Provision and adaptation requirements are more concerned with systems that can use an agreement to configure (or reconfigure) a service-provisioning architecture, and so presumably require an understanding of the semantics of the agreement.

Once again, these requirements cover much of the same ground as my own, although precision requirements are not emphasised, and measurement errors are not considered. Also, some of the requirements seem dubious, as they appear to be based on the assumption that earlier requirements can be met. For example, a need to specify third-party monitoring solutions is cited. However, as discussed in Chapter 5, this will only be feasible in some common scenarios if trusted monitoring solutions can first be implemented, and it is not yet clear that this is so. Again, a definitive specification for X-contracts is not yet available, so it remains to be seen how the authors will address some of these requirements.

Some discussion of expressiveness requirements is also provided in relation to the Web-Service Management Language (WSML) [111], in which it is observed that SLA conditions may be related to arbitrary external factors, and several example conditions are described to support this argument.

2.11 Summary

In this chapter I have presented a discussion of ASP, SLAs and the role that SLAs can play in an ASP scenario as a mechanism to mitigate the risks assumed by a client when choosing to use an outsourced service. I argue that these risks have been a major factor limiting the adoption of the ASP model of service provision. The discussion has established the assumptions upon which the work presented in this dissertation is based.

The discussion has highlighted the fact that an SLA could be used as a mechanism for providers to charge for their services, and that as a consequence of engaging in SLAs, providers will need to apply conditions to the behaviour of the client to prevent the client from exploiting the SLA.

I considered the kinds of conditions that parties to SLAs were likely to find most useful, and emphasised the importance of timeliness and reliability for clients, and throughput for service providers.

I also observed that in a typical ASP scenario more than one provider contributes to delivering the service to the client, and that therefore systems of multiple SLAs may be required to mitigate the client's risk, rather than assuming that all required conditions can be captured by a single SLA.

Proceeding from this discussion, I enumerated requirements for systems of SLAs, languages capable of expressing the SLAs in these systems, and the documents specifying these languages. In subsequent chapters in this dissertation, I use these requirements to justify the importance of contributions made to the theory supporting the specification of languages for SLAs, to inform the design and evaluation of a novel language for ASP SLAs, and to contribute to the demonstration of my thesis by providing a basis for comparison between the language-support I develop for ASP SLAs and that provided by pre-existing languages for the same purpose.

Chapter 3

Domain-specific languages for ASP SLAs

In this chapter I first introduce the main standards and prior work on the specification of Domain-Specific Languages (DSLs) upon which my efforts to produce a language for ASP SLAs depend. I then describe the first major contribution of this thesis, which is an set of recommendations concerning how these technologies should be combined to define a DSL for ASP SLAs, which will consequently exhibit good properties of understandability, precision and expressiveness. In Chapter 6, I describe in detail the design of a novel language for ASP SLAs, SLAng, according to these recommendations. However, in this chapter I discuss the recommendations at a theoretical level.

The recommendations, first described in [119], are that a language for ASP SLAs should:

1. be specified using a combination of the standard technical languages EMOF and OCL, described below, and natural-language descriptions;

2. be modelled using the model-denotational approach to provide both an abstract syntax for the language, and a precise description of the semantics of the language; and,

3. be abstract and extensible to best address the tradeoff required between restrictiveness and expressiveness in the ASP SLA domain.

EMOF, OCL and the model-denotational approach – itself a recommendation concerning the use of standards to define languages with precise semantics – are the contributions of other researchers. My contribution therefore consists of the following:

- the identification of these technologies as being particularly suitable for defining DSLs for SLAs, in comparison with other approaches to defining languages;

- the detailed explanation of how these technologies should be used to define a language for SLAs;

- the demonstration of the use of these technologies, in Chapter 6, to define an SLA language of realistic complexity; hence contributing an example in a novel domain to the hitherto quite small corpus of documented languages defined using this approach.

This chapter is structured as follows. In Section 3.1, I describe the standards and theory on which the approach is based. In Section 3.2, I demonstrate with examples how these technologies can be used to create a language for SLAs, and justify this approach according to its potential to deliver a useful

language for ASP SLAs. In Section 3.3, I compare the approach to other possible approaches to defining the syntax and semantics of formal languages. Finally, in Section 3.4, I summarise the material presented in this chapter.

3.1 Foundations of the approach

3.1.1 Object-oriented modelling

Object-oriented modelling languages permit the modelling of any subject, abstracted into a system of objects. Objects are conventionally regarded as things with measurable attributes, observable behaviours and relationships to other objects. They may be tangible, like a house or a person, or intangible, like an event, or the role somebody plays in a process. Object-oriented concepts are common in modern programming languages. However, object-oriented models differ from object-oriented programs, in that they can describe any system of objects in the real world, whereas object-oriented programs only describe systems of objects representing the structure and behaviour of computer programs.

Probably the most commonly used object-oriented modelling language is the Unified Modelling Language (UML) [81]. UML is a complicated language that allows a software system and its context to be modelled using a number of convenient abstractions. Different aspects of a system may be modelled separately, and the language facilities on which these views depend are to some extent independent of each other. The most commonly used subset of UML is the static structure part, also known as *class diagrams*.

Class diagrams, as a subset of the UML, are an object-oriented modelling language in their own right. They are not restricted to modelling software systems alone, as they are also intended to be used to model the context in which a software system operates. They are class-based, meaning that they model categories of objects, rather than representing unique objects directly. UML also allows the representation of unique objects in some other diagram types, but this is not as vital as it may seem, as a unique object can always be regarded as belonging to a class that contains only itself.

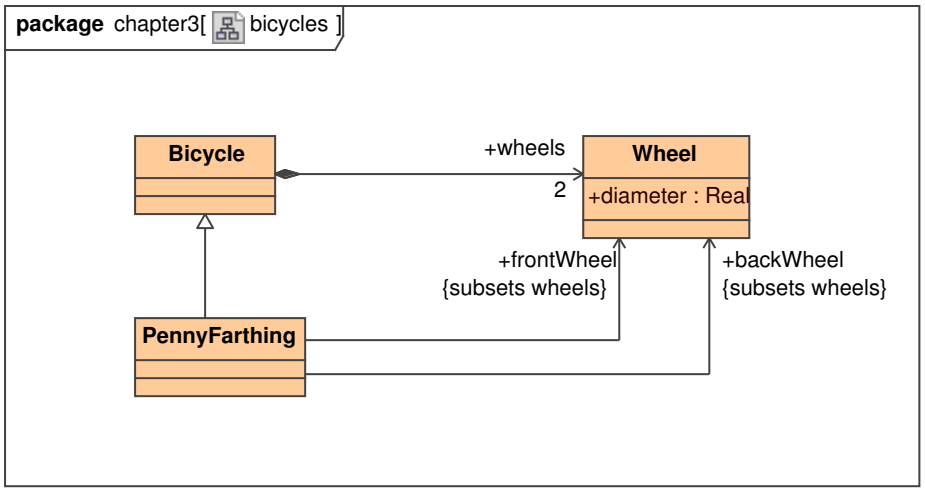


Figure 3.1: A UML model of bicycles

Figure 3.1 shows a simple class diagram. The diagram contains three classes, one representing all

the bicycles in the world, another representing all the wheels in the world, and a third representing all penny-farthings, an old-fashioned design of bicycle characterised by a very large front wheel.

Various relationships between these classes of things are shown. Bicycles have wheels, indicated by a composition relationship between the `Bicycle` class and the `Wheel` class. In the UML graphical syntax, a line decorated with a black diamond at one end represents a composition relationship, with objects of the class next to the diamond being the compositions, and objects of the related class the components. According to the standard interpretation of UML, components can only be part of one composition, and when the composition is destroyed, so is the component. This seems a fair model of the relationship between bicycles and their wheels. Numbers near the ends of these relationships represent multiplicity constraints: bicycles have two wheels.

Penny-farthings are a type of bicycle as indicated by the relationship between the `PennyFarthing` class and the `Bicycle` class, decorated with an arrowhead to indicate the superclass. All penny-farthings are bicycles, but not vice versa. Because the distinction will later be important, the relationships between penny-farthings and their front and back wheels are explicitly represented, using ordinary UML associations, which can be used to model any kind of relationship between two objects.

Wheels are modelled as having a diameter attribute, represented by a real number.

Note that I have only modelled a subset of all possible relationships between bicycles and wheels in general – another relationship might indicate whether a wheel could be used as a replacement on a bicycle. Neither have I modelled any other attributes that real bicycles and wheels have as a matter of course, such as weight, colour, or number of spokes. This highlights the fact that object-oriented models are abstractions of reality capturing only those aspects of interest to the modeller.

One of the strengths of UML class diagrams is that, given some preliminaries, it is reasonably easy to understand what they mean just by looking at them. However, if we wanted to be sure what the diagram in Figure 3.1 meant, we would have to refer to the UML language specification. This contains two pertinent sections: one explains how the symbols in diagrams map to abstract concepts in the language, e.g. boxes map to classes, and lines to relationships; and another defines the concepts, stating that a class is a category of objects or concepts possible in the real world, having the same relationships to other classes as shown in the model, and that beyond the structure of the model the dictionary definition of the name of the class is helpful in determining what real-world objects are being referred to.

In this sense, the UML language specification, with help from the dictionary, establishes a relationship between any diagram and all of the sets of objects or hypothetical situations that could be reasonably said to conform to the model that diagram depicts. This conformance relationship is represented in Figure 3.2. Three possible relationships between a model and a situation are shown. In the first, a teapot is very clearly found not to conform to the model of bicycles given earlier. According to the dictionary, a teapot doesn't resemble anything called a bicycle or a wheel. Moreover, the teapot considered alone doesn't have a structure similar to that in the model, in which the whole object includes two similar subcomponents. In the second situation, an actual bicycle is straightforwardly found to conform to the model. The bicycle itself conforms to the dictionary definition, as do both its wheels, and the wheels of

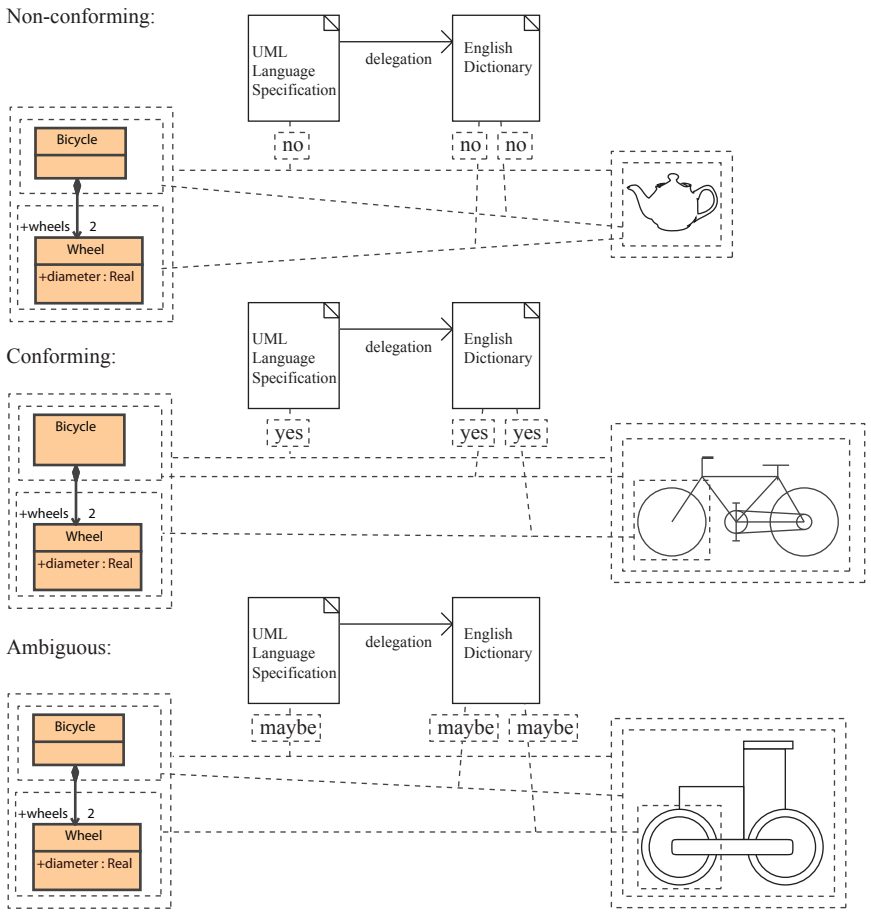


Figure 3.2: The UML specification, with help from the dictionary, determines what real-world objects conform to a model

the bicycle are in the the same relationship to the bicycle overall as specified by the diagram. Finally, a steamroller with two rollers is considered. Realistically, it is unlikely that a dictionary would leave much doubt as to whether a steamroller should be considered to be a bicycle, or whether a roller constitutes a wheel, but let us assume that the dictionary consulted includes rather generous definitions of these things. The model does not exclude the interpretation of the steamroller as conforming; certainly, the rollers of the steamroller are in the same relationship to the overall machine as the wheels of a bicycle. Therefore, the model may be said to be ambiguous in its relationship to steamrollers.

Adding more information to a model (refining it) can restrict the sets of objects that can be said to conform to the model, hence making the model more precise. It might be a matter of philosophical debate whether a two-rollered steamroller is a type of bicycle, but if it were important to do so, we could add detail to the model to explicitly either include or exclude steamrollers from consideration, for example by adding an extra class to represent steamrollers separately from bicycles.

UML class diagrams allow refinement in several ways. Additional classes, properties and relationships can be defined. Multiplicity constraints can be specified for relationships, as can the uniqueness and ordered-ness of members of those relationships. Properties can be added to classes, as can relationships. Some relationships between relationships can be specified, such as the subset relationship used

between the `frontWheel` and `wheels` properties in Figure 3.1. However, there are limitations to the expressive power of class diagrams, which mean that sometimes it is difficult or impossible to rule-out combinations of objects to which one does not wish to refer, or which would be illogical in the real world.

To address this problem, an auxiliary expression language may be used to specify invariants over classes specified in UML class diagrams. An invariant is a property of a class that always holds true. The language most commonly used for this purpose is the Object-Constraint Language (OCL).

For example, it would be extremely inconvenient to have to express the relationship between the size of the front and back wheels of a penny-farthing using class diagrams alone. However, in OCL it is easily expressed as an invariant over the class of penny-farthings:

```
frontWheel.diameter > backWheel.diameter
```

This is a very simple example of the use of OCL. However, much more sophisticated constraints can be written. OCL can also be used to express parametric calculations over objects of particular classes, known as side-effect-free operations. For example, we might define the following operations on the `Wheel` class to calculate the circumference, and the speed of the bicycle if the wheel is rotating at a given rate while the bicycle is in normal motion:

```
circumference() : Real = {

    let radius = self.diameter / 2
    in
    radius * radius * 3.1416
}

speed(revolutionsPerSecond : Real) : Real = {

    revolutionsPerSecond * circumference()
}
```

Because invariants can refer to side-effect-free operations, and the operations can refer to themselves recursively, making use of this facility renders the combination of UCL class diagrams and OCL version 2.0 Turing-complete, informally meaning that it is at least as expressive as the general-purpose programming languages, for example Java, in common use today. OCL can therefore express any property that can be checked by a conventional computer program, and this facility goes a very great distance towards enhancing UML class diagrams in their ability to discriminate between any two real-world situations. Models can therefore be specified with very high precision.

In summary, class diagrams, combined with OCL, offer two advantages to an author attempting to describe a situation clearly. They are potentially understandable, because they allow the direct description of the types and properties of objects which should be easily identifiable in the scenario being described, and need include only those aspects that are of interest. There are also no significant limitations on refining the diagrams to improve the precision with which they describe a scenario, at least in terms of properties checkable using conventional computers. These characteristics suggest the potential of these languages for defining SLAs, requirements of understandability and precision for which were identified in Chapter 2.

3.1.2 The Object Management Group (OMG) and the Model-Driven Architecture (MDA)

In this section I briefly describe the Object Management Group (OMG), a standardisation organisation, and its Model-Driven Architecture (MDA) initiative. The history of this organisation is relevant to features of standards upon which I later depend, and I also discuss certain contributions of this work with reference to the MDA, in later chapters.

UML is a standard of the OMG. The OMG is a standardisation organisation whose membership consists of industrial and academic organisations, and whose stated purpose is to standardise technology that can assist in the integration of distributed Enterprise Information Systems (EISs).

UML was originally the combined product of several independent research efforts to develop general-purpose object-oriented analysis and design languages. The objectives of this work were to improve the quality of domain analyses informing the requirements of software systems systems, reduce the cost of developing software systems in object-oriented programming languages, and improve traceability between analysis and design artifacts in such developments. To achieve widespread acceptance the UML clearly required standardisation, but why did the OMG, with its mission to integrate distributed systems find UML an attractive prospect for adoption?

In its early history the OMG standardised the Common Object-Request Broker Architecture (CORBA), a sophisticated programming-language-independent middleware standard. The advantages of middleware to integrating distributed computer systems are obvious: middleware establishes standard communication protocols enabling systems to implement electronic services and thereby communicate with each other; it also provides reusable libraries, services, and generative programming tools that reduce the cost of implementing a new electronic service or refactoring a legacy system into an electronic service. However, it was found that middleware alone did not address all of the problems encountered when integrating EISs, and a modelling language such as UML had a role to play.

Perhaps the strongest original motivation for the adoption of UML as a standard by the OMG was that it was perceived to offer a solution to the common problem of reconciling the interfaces and data models of two EISs. EISs may have several interfaces offering many different operations. The data-model of such systems is often implicit, but is important because it governs the encoding and meaning of data-structures and parameters passed to operations. The integration of two systems requires at least that their interfaces to be understood, and the implementation of any translations required between the data models of the two systems. UML offered the prospect of a common object-oriented language in which interfaces and data-models could be defined. These models could then be made available in online meta-data repositories, to allow services to be discovered and integrated by automated tools.

This ambition has never been fully realised, perhaps due to the extremely difficult theoretical challenges posed by reasoning with UML models, and also the fact that repository-based approaches to automatically integrating systems tend to neglect important business considerations, such as the possible need to enter into SLAs, for example. However, the idea has its continuation in the OMG's efforts to standardise 'Domain Specifications'. These standards define standard electronic-services interfaces in CORBA's Interface Definition Language (IDL) with supporting object-oriented models of data, for

various application domains such as gene-expression data [84] or product life-cycle management [96]. By planning support for these standards in their EISs, organisations can reduce the expected costs of integration with other EISs that also support the profiles.

A later and stronger argument for UML's usefulness in the integration of EISs was presented when the OMG announced its MDA initiative [78]. The key recommendations of the MDA approach are that systems be developed primarily using models (usually UML models); and that these models should be developed using a process of refinement whereby details relating to the application domain are added earlier, and then details related to the chosen implementation technologies are added later, and preferably automatically. The benefits of these prescriptions are two-fold: first, the availability of the early models means that an application can more easily be re-implemented using different technology if this becomes desirable at a later date; second, because refinement is automated as much as possible from early models, the cost of implementation is reduced once the application domain has been modelled.

The MDA was proposed to address a perceived deficiency of CORBA, which was that in the years following its standardisation a number of competitive middleware standards emerged, most significantly Enterprise Java-Beans (EJBs) [131] and webservices [145]. If choosing to support standard middleware can no longer be relied upon as a strategy to insure an EIS against future integration costs, then it is necessary to plan the implementation of an EIS with a view to reducing the future costs of re-engineering to support a new middleware when required. It is anticipated that developing a system using an MDA approach will contribute to reducing these costs, as any models independent of middleware technology can be used as the basis for a new process of implementation-by-refinement targeting a new middleware. Efforts are also underway to define approaches and technology for the extraction of technology-independent models from legacy systems that were not originally developed according to an MDA process.

When the OMG adopted UML it consisted of a collection of diagrammatic notations, the structure and meaning of which were described informally. To be useful for expressing meta-data in online repositories, and as the input for tools able to automatically manipulate models in MDA developments, UML required a degree of formalisation. Efforts to achieve this have resulted in several theoretical advances in the definition of modelling languages, and the publication by the OMG of a family of standards of use in defining domain-specific languages, introduced in the next section.

3.1.3 The syntax of modelling languages

An *abstract syntax* is a description of a language that is 'analytic, rather than synthetic' [59], in that it establishes a set of rules whereby a statement may be regarded as conforming to a grammar rather than a set of rules whereby smaller statements may be combined into larger statements, as is characteristic of generative formal grammars [73].

An object-oriented abstract syntax is an object-oriented model of the information contained in the statements expressible in some language. A very simple abstract-syntax for a language for cataloguing the contents of warehouses containing bicycles is shown in Figure 3.3. A catalogue consists of an identifier for the warehouse being described, and also a number of product descriptions, describing the

contents of the warehouse. A product description includes the number of the shelf on which the product is stored, and some more information about the product that is dependent on the type of the product. In this example, I have only provided syntax for describing bicycles, and that only in the very limited sense of being able to say whether the bicycle is a penny-farthing, or not.

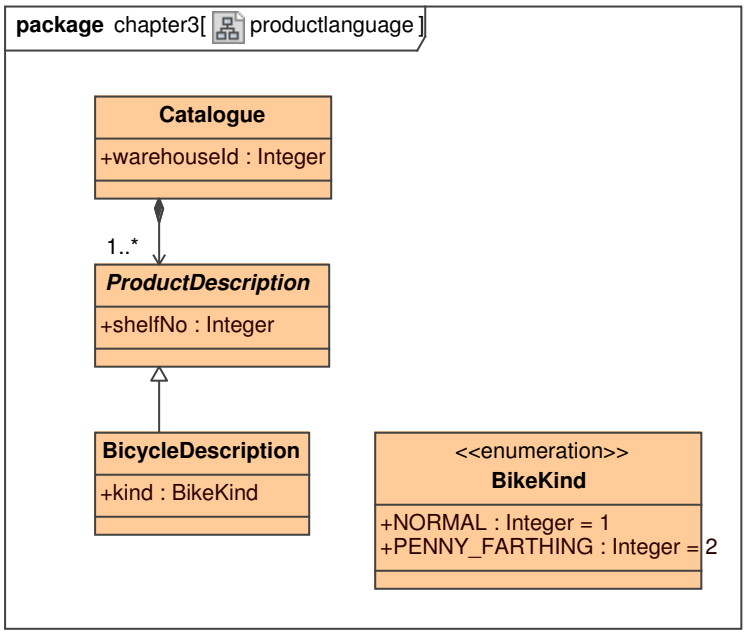


Figure 3.3: An abstract syntax for a simple language for cataloging warehouses

The word ‘abstract’ in the term ‘abstract syntax’ refers to the fact that such grammars model what a language expresses, but don’t say how this information must be represented. Note that an object-oriented abstract syntax may contain both concrete and abstract classes (indicated by an italic class-name). Concrete classes represent identifiable pieces of information in a language statement. Abstract classes, such as the `ProductDescription` class in my example, represent categories of statement elements with common characteristics. In terms of the conformance relationship defined by UML, described in Section 3.1.1, conformance to a concrete class can be determined by comparing a real-world object with the properties of that class (its name, attributes and relationships). On the other hand, conformance to an abstract class can only be determined by comparing a real-world object to a concrete subclass of the abstract class. This is because abstract classes have some characteristics that are not fully defined. Product descriptions, in the example, are expected to somehow represent a product, but this will necessarily include more information than merely the product’s location. However, the structure of this extra information will depend on the type of the product, so will not be a feature of all statements in the category of product descriptions.

The notion of using an object-oriented abstract syntax for a modelling language was originally introduced to address a pair of deficiencies in early versions of the UML [47], and was made possible by the realisation that UML class diagrams are an appropriate formalism for modelling the structure of the UML language as a whole, resulting in a recursive definition of the language.

The problems addressed were as follows: first, although the graphical syntax of UML was standardised, there was no standard way to either exchange models between tools implemented by different vendors (a common benefit of standardisation), or manipulate models programmatically via the interface to a meta-data repository (an early objective of the OMG's); second, the UML is based on the philosophy that the best way to model a system is from a collection of viewpoints, each capturing an aspect of the system. For example, one viewpoint may describe the static structure of a computer system, another its behaviour, and yet another the way that it is packaged and deployed onto hardware resources. According to this philosophy, the viewpoints are expressed in separate diagrams, each with a specialised visual language. However, this raises the possibility that inconsistencies between the views may be introduced by the modelling process. These inconsistencies could eventually result in flaws in the developed system, so it was desirable to provide a mechanism whereby they could be either detected or prevented.

The provision of an abstract syntax for UML addressed both of these problems to a significant extent. By regarding a model as a structure of objects two benefits accrued: first, well-understood techniques for encoding objects as documents could be borrowed from the object-oriented programming world, and standardised to create a document-interchange format, and a standard document-model that could be manipulated via programmatic interfaces; second, in a given project, the several diagrams produced in UML could now be regarded as merely projections of a coherent underlying object-oriented model of the system. This was analogous to another object-oriented programming technology, the Model-View-Controller (MVC) pattern [48], used to maintain consistency between several user-interface components by maintaining the data to be displayed (the model) separately from the visual representation of the components (the views). The problem of maintaining consistency between the diagrams in a UML development is hence reduced to the problem of maintaining consistency within the model, and consistency between the model and the diagrams.

OCL was also introduced as an early refinement to UML, first as an optional part of the UML specification, and later as an independent standard. One of the most conspicuous early uses of OCL was to define additional consistency constraints and side-effect-free operations on the abstract syntax of UML as described in the UML language specification. However, since its introduction, OCL has also has facilities that assist in modelling the dynamic behaviour of software in UML models, by defining pre- and post-conditions for operations on classes.

Motivated by the desire to standardise a CORBA service for the provision of meta-data in EISs, the OMG defined the Meta-Object Facility (MOF) standard, common services in CORBA being called 'facilities'. This standard reproduced the static-structure (class diagrams) part of the UML standard, and described a mapping from instances of this model (referred to as the MOF model) to sets of interfaces defined in the CORBA Interface-Definition Language (IDL). This allowed for the possibility of defining meta-data structures as instances of the MOF-model, and then automatically generating a CORBA service that could store, retrieve and edit data conforming to these structures.

Following the standardisation of the MOF, the OMG have adopted the policy that future language standards should have abstract syntaxes defined using the MOF model. Since the MOF model essentially

defines an abstract-syntax for an object-oriented modelling language, in most respects indistinguishable from UML class diagrams, the term 'MOF' is often used to refer to the language that the specification defines, and I use the term in this sense below. Moreover, UML class diagrams are commonly used to represent MOF models.

Since UML 1.1, MOF has been used to define the UML. MOF also has an abstract syntax, similar to the abstract syntax for UML class diagrams, which is recursively defined as an instance of the MOF model. Because MOF defines the UML, which is a modelling language, instances of the MOF model are frequently referred to as 'meta-models'. Most meta-models in fact define the abstract-syntax of some language, which may or may not be a modelling language.

This system of standards is described in the UML specification as a 'four-level meta-modelling architecture', as shown in Figure 3.4. Objects at each level of the architecture represent a theory concerning the structure of objects at the layer below. At level M0 are real-world objects. These are described by UML models at level M1. The meta-model of UML is at M2, an instance of the MOF model at level M3. This architectural model has some serious logical inconsistencies: the MOF, as an instance of itself could plausibly also appear at level M2, and all levels above level M3; any language specification arguably describes not only its own structure, but also its meaning, and therefore governs two meta-layers beneath itself, rather than one; finally, models and language specifications are objects in the real-world, so can equally easily be argued to exist at level M0. Nevertheless, the model is helpful in understanding the relationship between the standards.

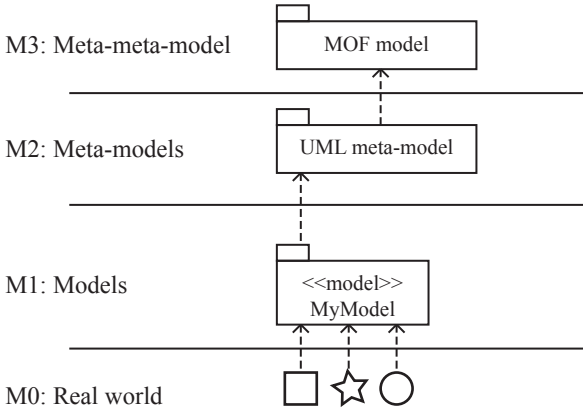


Figure 3.4: The four-level meta-modelling architecture, as defined in the introduction to the UML 2.0 standard

The OMG's solution to providing a document interchange standard for UML is the XML Meta-data Interchange (XMI) standard. Similarly to the MOF standard, the XMI standard provides a mapping from a MOF model, however in XMI's case it is to an XML grammar. In earlier versions of the standard this was a Document-Type Definition (DTD) [142]. Currently it is an XML schema [24]. Constructs in an XMI grammar correspond to the concrete classes in the abstract syntax from which it is generated. The advantage of this approach is that any language with a sufficiently refined abstract syntax defined using the MOF is also implicitly defining at least one concrete syntax. For example, a sample statement in the warehouse-cataloguing language shown in Figure 3.1, encoded in XMI is:

```
<Thesis:Catalogue warehouseId="0" xmi.id="mofid:3040">
<Thesis:Catalogue.productDescription>
<Thesis:ProductDescription xmi.idref="mofid:3043"/>
<Thesis:ProductDescription xmi.idref="mofid:3041"/>
</Thesis:Catalogue.productDescription>
</Thesis:Catalogue>
<Thesis:BicycleDescription
    kind="NORMAL" shelfNo="1" xmi.id="mofid:3041"/>
<Thesis:BicycleDescription
    kind="PENNY_FARTHING" shelfNo="2" xmi.id="mofid:3043"/>
```

Unfortunately, the grammars produced by the XMI standard are not particularly easy to write by hand. This means that in many cases, developers wishing to use UML or another languages have little choice but to use graphical editors, which may be expensive to acquire or develop, and are difficult to integrate into automated software development processes. To address these issues the OMG provided yet another standard, the Human-Usable Textual Notation (HUTN), which like XMI also maps a MOF model to a grammar, but in this case it is defined in Backus-Naur form, and results in a syntax for a language that is more similar to a block-structured programming language like Java.

The HUTN version of the above equation-language statement is as follows:

```
Catalogue() {

    productDescription = {

        BicycleDescription() {

            shelfNo = 1;
            kind = NORMAL
        },

        BicycleDescription() {

            shelfNo = 2;
            kind = "PENNY_FARTHING"
        }
    }
}
```

In the most recent versions of the UML and MOF standards, the commonality between the language defined by the MOF-model and class diagrams has been acknowledged. The extremely large meta-model for UML version 2 and later has been subdivided into a number of reusable packages, and MOF version 2 is defined with reference to the same packages that underlie UML class diagrams. However, the introduction of novel package reuse mechanisms has complicated the MOF standard considerably. Hence the OMG has subdivided the MOF standard into two sub-standards, or conformance levels. These are the CMOF, or Complete-MOF, and EMOF, or Essential-MOF. EMOF is considerable simpler in structure than the CMOF, and has seen much greater adoption by tool manufacturers. The version of the EMOF meta-model [86] implemented by the UCL MDA tools (described in the next chapter) is depicted in Figure 3.5.

Subsequent to the standardisation of MOF, the Java-Community Process standardised a very sim-

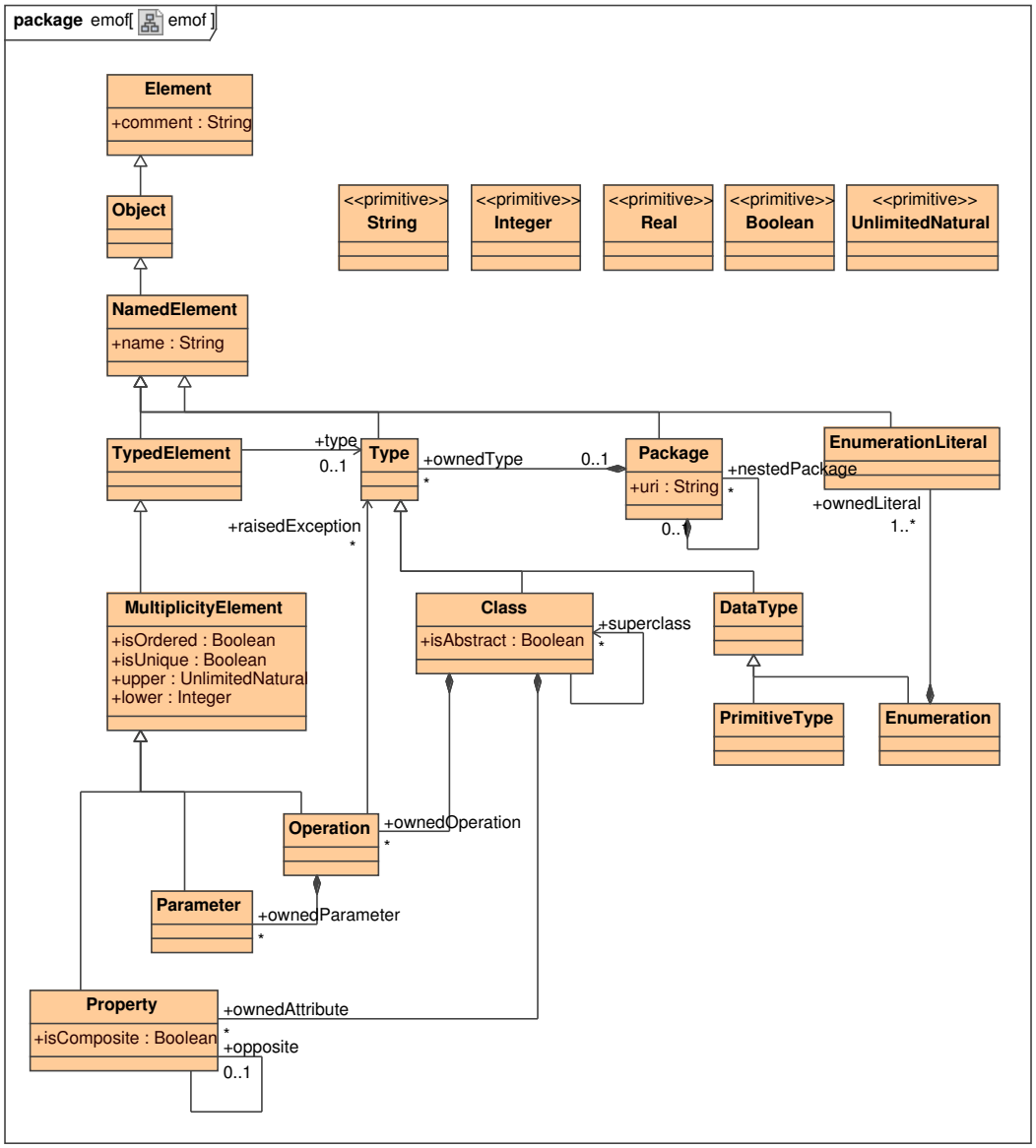


Figure 3.5: The EMOF meta-model from the draft MOF version 2.0 core proposal

ilar technology for generating Java interfaces from MOF models, the Java Meta-data Interface (JMI) specification [129]. This standard specifies the structure and behaviour of interfaces for accessing instances of a MOF model within a Java program. Several implementations of the standard are freely available, and these include the ability to also generate classes implementing the interfaces to provide an in-memory model repository. Various UML editors currently rely on a JMI implementation of the UML meta-model to store the working copy of any models they manipulate. Probably the most popular current implementation of the JMI standard is the Eclipse Meta-data Facility (EMF) which relies on a simple meta-modelling language very similar to EMOF [21].

3.1.4 The semantics of modelling languages

In Section 3.1.1 I have described the way in which the UML language specification defines a conformance relationship between UML models and real or hypothetical sets of objects. This relationship

constitutes the semantics, or meaning, of the language. In this section, I discuss approaches to describing the semantics of such languages.

It is perhaps a surprising observation that English is currently the state-of-the-art language for defining full modelling languages, such as UML and MOF, in OMG standards. These languages have an abstract syntax, several concrete syntaxes, and semantics. The abstract syntaxes are structured as MOF models. However, they are not generally defined by a concrete artifact expressed in the MOF language, such as a MOF XMI file. Instead they are described in a specification distributed in the Portable Document Format (PDF). Although PDFs are machine-readable for the purposes of displaying and printing a human-readable document, they do not allow the inspection and manipulation of the meta-models of these languages as XMI would. In the UML version 1.5 specification, the structure of the meta-model was definitely established using a combination of class diagrams and supporting natural language descriptions, in English [87]. The English descriptions are necessary to disambiguate the diagrams, which occasionally suppress details and which systematically omit any definition of their context. In UML 2.0 the use of diagrams is deprecated to purely informal [81]. The structure of the meta-model is definitively established in 'formal concept definitions' associated with elements in the abstract syntax, and structured according to the features of the MOF type of the element being defined.

The semantics of MOF and UML are described to a large extent by attaching natural language descriptions to the elements in the abstract syntax in the PDF language specification documents. Part of the semantics of these languages is also defined by the structure of their meta-models, and by the English language names used to name the elements in these meta-models. These names and the relationships between the elements evoke real world scenarios (involving objects and classes, activities etc.), and a reader of these specifications can fairly interpret the meta-models as referring to these. However, the semantics are are finally and definitively established by the English language comments associated with these elements. In the case of UML 2, these are included in the formal concept definitions for each element of the abstract syntax.

Structuring natural language semantics according to the meta-model of the language being defined results in language documentation that is at least accessible and complete. In the case of MOF and UML, I somewhat controversially argue that the semantics are also reasonably precise. Consider the meta-model for EMOF shown in Figure 3.5: the abstract syntax of the language has very much the same structure as the domain of real world objects being described. The model of EMOF could instead be regarded as an abstract model of the real world as a continuum of classes of objects (ignoring the package mechanism, which doesn't have a semantic interpretation). The semantic relationship between statements in the language and the domain of the language is hence a reasonably unambiguous one-to-many mapping: a class in a model describes any real-world class of objects that has the same structure and relationships as the class in the model, taking into account the interpretation of the natural-language components of the model (conversely, any set of objects, matching the structure and natural-language elements of a model, conforms to it). Given that this is the case, the descriptions of the meaning of each element of the MOF model can be quite simple and unambiguous.

Furthermore, any model, including meta-models, which may be regarded as models of languages, must eventually be described using natural language if its correspondence to its subject is to be understood. Even if the meaning of a model is described using a mapping to another formalism, that formalism must eventually benefit from a natural language description, or it would remain forever an unintelligible mathematical structure. The relatively similarity between MOF models and their subject makes an immediate description of their semantics in natural language an appropriate choice. This is also true for some parts of the UML, obviously including the class language that it has in common with the MOF.

A purely natural-language approach is not always appropriate though. UML includes two 'lightweight extension mechanisms', called stereotypes and tagged values. These are essentially syntactic constructs without predefined semantics. Stereotypes allow the labelling of any UML syntax element with a string. Tagged-values allow the same labelling with name-value pairs. A stereotype is used in Figure 3.3 to label the class `OperationKind` as an enumeration, because UML requires extension to support enumerated types.

Stereotypes and tagged-values must be declared in a UML model before they can be used, and a collection of stereotype and tagged-value declarations may be packaged into a reusable language 'extension' known as a profile. Profiles are commonly used to mix some domain-specific expressive capabilities into the UML, and a number of profiles have been standardised, such as the Profile for Schedulability, Performance and Time Specification [89], which allows quantitative performance information to be included in models, and the Enterprise Distributed-Object Computing Profile [82], which allows the inclusion of technical information specific to the implementation of EISs using middleware.

The incapacity of profiles to modify the meta-model of the UML to reflect their own domain of interest has led to the common practice of providing a domain model, with reference to which the semantics of the profile are defined. A stereotyped element in a UML model is taken to imply the existence of an instance of a class in the domain model, hence determining the semantics of the stereotype. Tagged values specify the values of properties of these semantic objects. Domain models in profiles are often defined using MOF, to permit their alternative use as an independent domain-specific language.

The style of semantic definition for meta-models, now widely referred to as 'model-denotational', was pioneered by the Precise UML group [23], and employed in its submissions to the UML 2 standardisation effort, extends this notion by formalising the relationship between syntactic elements and domain model elements using standard meta-model relationships and constraints. In practice the syntactic model and the domain model form a joint meta-model for the language, in which the notion that all meta-model elements are elements of the abstract syntax of a language is dropped.

The principle advantage of this type of definition is that there does not need to be a simple correspondence between syntactic elements and the notions that underlie them to permit a precise description of the semantics of a meta-model. Because the domain elements are atomic and well-understood, they may be documented simply and unambiguously using natural language. However, the complex relationship they bear to the syntax of the language is defined formally by the associations and constraints in the model.

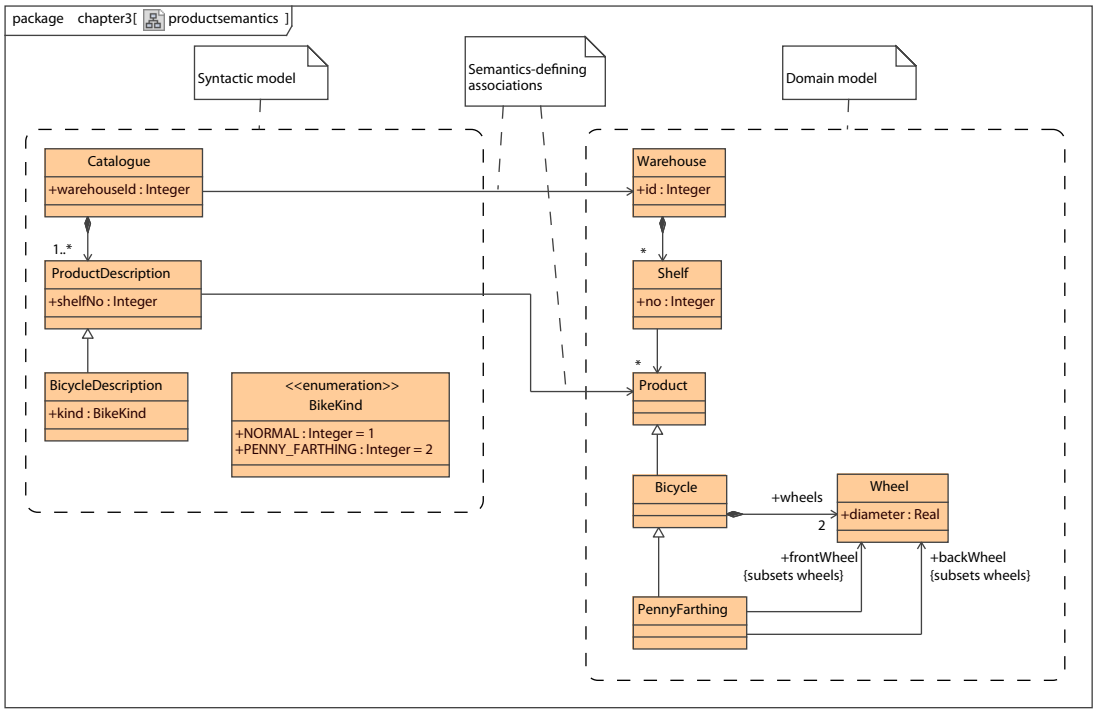


Figure 3.6: Model-denotational semantics for the warehouse catalogue language

The meta-model of a language specified using the model-denotational approach is depicted in Figure 3.6, in which the syntactic model of the warehouse-catalogue language, described in Section 3.1.3 has been associated with the model of bicycles, originally introduced in Section 3.1.1. This domain model has been expanded to add the notions of warehouses and shelves, and a generalisation of bicycles as products that may occupy shelves. To complete the definition of the language, the following invariants are needed:

On the `Catalogue` class:

```
warehouse.id = warehouseId
and
productDescription->forall(p : ProductDescription |

    warehouse.shelf->exists(
        no = p.shelfNo
        and
        product->includes(p.product)
    )
)
```

This establishes that to conform to a catalogue, a warehouse must have the same identifying number as listed in the catalogue, shelves corresponding to the shelves listed in product descriptions, and products on those shelves matching the product descriptions.

On the `BicycleDescription` class:

```
product.oclIsKindOf(Bicycle)
and
kind = BikeKind."PENNY_FARTHING"
```

```
    implies product.oclIsKindOf(PennyFarthing)
and
not (kind = BikeKind."PENNY_FARTHING")
    implies not product.oclIsKindOf(PennyFarthing)
```

This establishes what it means for a bicycle to conform to a description of a bicycle in the catalogue. Trivially, if the catalogue says that the bicycle is a Penny-Farthing, then the bicycle must be a Penny-Farthing, and not otherwise.

If this language were to be put into use, the various classes in the language specification should also be commented in natural language to definitively establish their correspondence to real-world objects, in some definitive language specification document.

Note that the structure of statements in the language (i.e. catalogues of warehouse statements), is not identical to the structure of the semantic domain. For example, the product description of a bicycle contains no information pertaining to the number of wheels that a bicycle has, or the relationship between the diameters of the wheels on a penny-farthing. However, these details are provided by the domain model, in a precise manner. This means that these details do not need to be described informally in comments associated with the `BicycleDescription` class. Neither does the `BicycleDescription` class need to be extended with additional structure to allow these details to be specified by the author of a bicycle description, which would be redundant, as they would need to be present in all bicycle descriptions (assuming these details are genuinely of relevance to the application of the catalogue language).

The approach was not adopted in UML 2, perhaps due to the perceived lack of a strong need to define an intermediate semantic model to describe UML.

The model denotational approach has found some adoption in standards. The OCL 2 specification provides such semantics [88]. The semantics of OCL 2 are defined in terms of expression evaluation events which are somewhat different in structure to the underlying expressions (loop evaluations are rolled out, for example). Also, the style is used to define an 'abstract semantics' for CMOF in the MOF 2 specification [79].

A number of attempts have been made to introduce more traditional styles of semantic definition of languages with MOF-defined abstract syntaxes. A popular option has been to employ a traditional mathematical approach based on logic and set theory. A 'formal' semantics has been proposed for OCL, for example, although it is not definitive, nor is equivalence with the definitive model denotational semantics proven [109]. Alternative approaches to defining semantics for modelling language are discussed further in Section 3.3.2.

3.2 Abstract, extensible, domain-specific languages for SLAs

3.2.1 Modelling SLAs

The conformance relationship between real-world situations and class diagrams described in the Section 3.1.1 suggests a way in which SLAs could be specified. The agreed behaviour of all parties and the service in a service-provisioning scenario could be modelled using class diagrams. This model could be included in an SLA, or even constitute the SLA, with the following stipulation to the parties: if your behaviour and that of the service conforms to the model, then you are complying with the SLA, otherwise

you are violating the SLA.

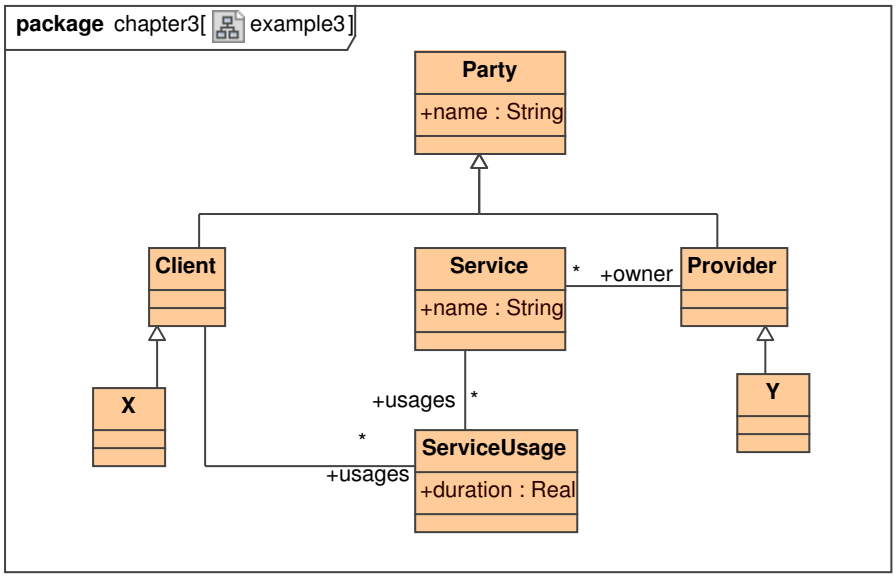


Figure 3.7: A UML model of a service-provisioning relationship

For example, consider the service scenario modelled in Figure 3.7. Let us assume that the hypothetical parties X and Y wish to enter into a relationship with respect to a service Z. The following additional invariants are required to fully constrain the relationship between X and Y, expressed using OCL.

On class X:

```
name = 'X'
```

On class Y:

```
name = 'Y'
and
service->exists(name = 'Z')
```

And on the class of services:

```
name = 'Z'
and
owner.name = 'Y'
implies
usages->forall(u : ServiceUsage |

    u.client.name = 'X'
    implies
    u.duration < 10
)
```

This model therefore represents an agreement between X and Y, such that when Y provides its service Z to X, it takes less than 10 seconds to complete, each time it is used. Of course, the model as proposed is still very ambiguous. It is not clear, for example, how the duration of a service usage should

be measured, or in what units recorded. These problems could be addressed by further refinements to the model, as discussed in Section 3.1.1.

Adopting a combination of UML and OCL is attractive for specifying SLAs because these languages together meet a number of requirements of SLA languages quite well. They are highly expressive, and so should be able to capture any requirement that the client has for the behaviour of the service, or any requirements that the parties have regarding the behaviour of their peers. They are fairly understandable, and should be at least somewhat familiar to anybody with the expertise to specify an SLA for an application service. They also have the potential to express precise SLAs, are supported by a range of tools, improving their ease of use and usefulness in software-engineering activities, and benefit from a machine-readable syntax thanks to the XMI standard.

However, the principal deficiency of this combination of languages is that although it is capable of expressing good SLAs, it provides no real support for doing so. Almost all of the details concerning the scenario to which the SLA applies must be completely specified in the SLA, despite the fact that it might have a lot in common with scenarios for which previous SLAs have been specified. The languages are also not restrictive, so it is very easy to express SLAs that are ambiguous, as in the above example, or that fail to correctly capture the intent of the parties with respect to the agreement. Verifying that an SLA encodes the required conditions, and does so in a way that is unambiguous, monitorable and difficult to exploit, will always be be the sole responsibility of the author. This may increase the cost of preparing an SLA, because of the extra effort required to validate that these properties hold, or result in residual flaws in the SLA.

In the next section I look at how these deficiencies can be addressed without losing the benefits of object-oriented modelling for expressing SLAs.

3.2.2 Reusable models of SLAs

A common approach to providing reusable domain-specific facilities in a general-purpose language (or a general-purpose programming language) is to provide libraries. For example, legal contracts often reuse boiler-plate text; the Java programming language includes an extensive standard library to support many common programming tasks, such as providing user-interfaces, processing documents, and interacting over a network.

Like Java, UML includes a package mechanism that can be used to hierarchically subdivide models. This allows models, and parts of models, that have developed separately, to be combined without introducing ambiguities caused by name clashes. The package mechanism could be used to encapsulate and redistribute reusable models of service scenarios, thereby reducing the effort required to apply the approach described in the previous section. Ignoring for a moment its many faults, let us see how this would work using the example service model previously described.

It would not be sensible to redistribute the details of the relationship between X and Y, which may be private. The model will therefore have to be parameterised somehow, with the parts that are common to several relationships redistributed, and the parts that are specific to a single relationship elided, or referred to only in abstract.

Let us assume that the redistributed model of service usage becomes very commonly used, and hence well known. If this happened, then SLAs relying on the model would not need to redistribute the model themselves, they could merely refer to it. An SLA itself would only consist of any extensions to the model required to make it specific to a particular service-provisioning relationship. Because the model was well known, and presumably also easy to obtain, anybody receiving such an SLA would know what it meant.

One possibility for parameterising the model is therefore to locate all of the parameters in a single element, modelling an SLA that contains only the information specific to a particular relationship. Figure 3.8 shows the service-provision model from the previous section adapted in this way.

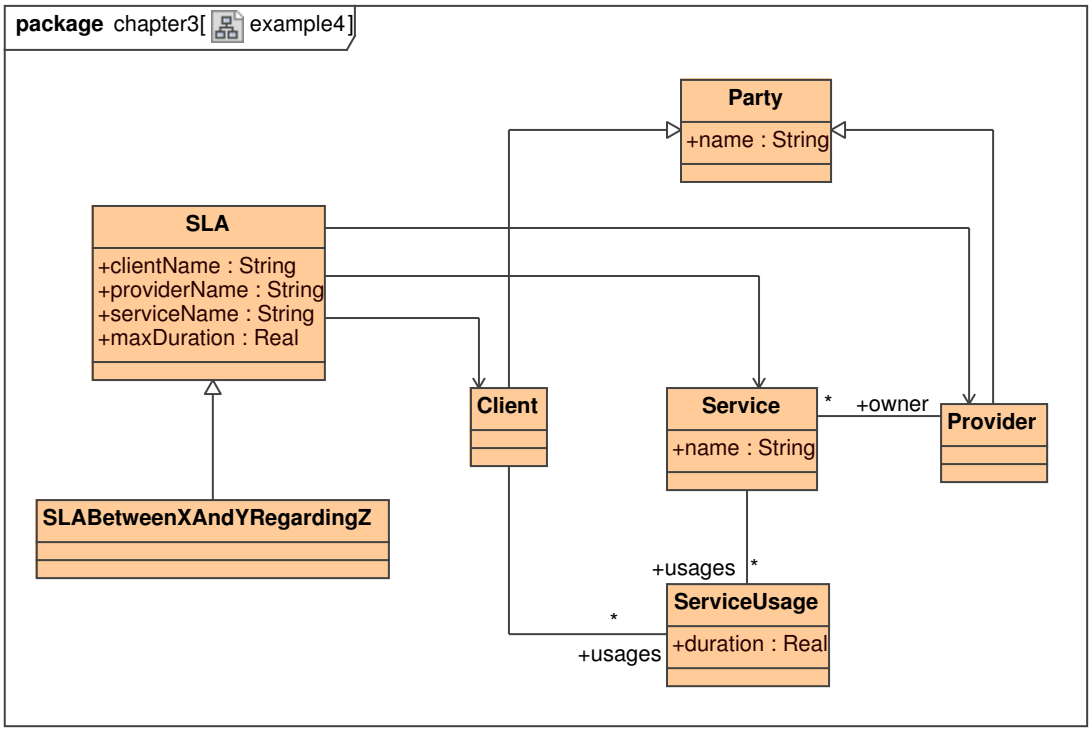


Figure 3.8: A more reusable UML model of a service-provisioning relationship

The invariants governing the relationship can now be moved into the `SLA` class. The first establishes that certain information in an SLA defines its relationship to the service provisioning scenario to which it will apply.

```
client.name = clientName
and
provider.name = providerName
and
service.name = serviceName
and
service.owner = provider
```

The second ensures the SLA contains a meaningful limit on the duration of service usages:

```
maxDuration > 0.0
```

The third governs the relationship:

```
service.usages->forall(u : ServiceUsage |

    u.client.name = clientName
    implies
    u.duration < maxDuration
)
```

By way of example, the SLA between parties X and Y has been re-implemented, by extending the `SLA` class. The parameters for the SLA are now constrained in the class `SLABetweenXAndY-RegardingZ` using another invariant:

```
clientName = 'X'
and
providerName = 'Y'
and
serviceName = 'Z'
and
maxDuration = 10
```

The `SLA` class in the modified scenario can be regarded as more than just a convenient point of extension for parameterising the scenario. The model, consisting of the type structures shown in Figure 3.8 and the invariants listed above, represents a description to which exactly one (non-empty) situation in the real world is expected to conform - if no scenarios conform to the model, then it either contains an error or the SLA has been violated; if multiple scenarios conform to the model, then it must be ambiguous in identifying the parties or service expected to be involved in the relationship. Moreover, the parties to the SLA have agreed that the service-provision scenario should in principle conform to this model. Therefore, the inclusion of the class `SLA` represents the expectation that a concrete SLA document will exist in the scenario, and will have particular contents, in this case defining the participants, identifying a service provided by the provider to the client, and a limit on the duration of the service usages for the service.

The `SLA` class therefore represents the extension of the model that is needed to specify details of the relationship specific to a particular relationship. In the example above this is the definition of the class `SLABetweenXAndYRegardingZ`, which provides all of the requisite information in its invariant, thereby conforming to the definition of the `SLA` class both as a subclass and an instance. The model does not say how an SLA needs to represent this information, merely that it must convey it somehow. In this sense, the model has defined an abstract syntax for these types of SLA. The definition of the class `SLABetweenXAndYRegardingZ` is therefore the SLA in this case, and the subset of OCL used in its invariant can be regarded as conforming to this abstract syntax.

The model has also defined what the SLA, in this case the definition of the class `SLABetweenX-AndYRegardingZ`, means. In any situation where the SLA is not violated, there will also be a service and participants meeting the constraints as parameterised by the SLA. The meaning of the SLA is defined by the model, and in a situation conforming to the model, the existence of an SLA necessarily implies the existence of the parties and service associated with the SLA. We can say that a particular SLA denotes a particular service situation, and the model hence constitutes a model-denotational description of the semantics of the language.

This approach to supporting the specification of SLAs is an improvement on the approach suggested in the previous section in that it has improved the power of the approach, in the sense that having obtained a reusable model for a class of SLAs, less work needs to be done to create each new SLA within that class. The restrictiveness of the approach has also been improved, because the structure of the SLAs is tightly defined by the `SLA` class, and constraints, such as that requiring the maximum duration for a usage to be non-negative, have been included. These features combine to make it more difficult to define a bad SLA based on the model.

The principle drawback to this approach is that SLAs must be specified as classes extending a core element. This is not as restrictive as having a dedicated syntax in which SLAs can be written. However, the example in this section has highlighted the fact that producing a reusable model for SLAs can be equivalent to defining an SLA language with model-denotational semantics. In the next section, I describe this approach in detail, and discuss its advantages.

3.2.3 Recommendations for developing languages for ASP SLAs

I now present the principle contribution of this chapter, which is to argue that adhering to the following three recommendations when defining a language for ASP SLAs will tend to result in a language that meets the requirements described in Section 2.8 well:

1. a language for ASP SLAs should be specified using a combination of the technical languages EMOF and OCL, and natural-language descriptions;

2. a language for ASP SLAs should be modelled using the model-denotational approach to provide both an abstract-syntax for the language, and a precise description of the semantics of the language;

3. a language for ASP SLAs should be abstract and extensible to best address the tradeoff required between restrictiveness and expressiveness in the ASP SLA domain.

The first and second recommendations represent a minor philosophical adjustment to the approach presented in the previous section.

Continuing the argument of the previous section, a specification of an ASP SLA language that consists of an abstract syntax plus a model-denotational definition of semantics is equivalent to a model of a set of service scenarios in which some statements (the SLAs) have an impact on what is considered to legitimate behaviour from a scenario (i.e. that behaviour required for the scenario to conform to the model). The principal difference between a language defined according to my recommendations, and the type of model described in the previous section is that the language specification will be expressed using EMOF rather than UML. This represents an attempt to conform to the standard style of the OMG when defining languages, and to benefit from compatibility with standards such as XMI, HUTN, and JMI which provide concrete syntaxes for the language, and offer the possibility of tool support.

This differences is essentially trivial. UML class diagrams and EMOF have very similar expressive powers. Moreover, the discussion in the previous section supports the assertion made in Section 3.1.1 that there is little semantic difference between specifying a unique object, and specifying a class which can contain only a unique object. Because the distinction between models and meta-models, or languages

and statements, can be blurred in this fashion, it becomes clear that distributing an EMOF model of a language expressed in the model-denotational style is essentially equivalent to distributing a reusable object-oriented model in a general-purpose modelling language.

The differences have an important practical implication however. Regarding a redistributed model of a scenario as a language specification introduces a categorical distinction into the activities required to produce a concrete SLA.

According to the approach described in the previous section, to produce an SLA, it is necessary to obtain an appropriate reusable core model and then extend it until it precisely describe describes the scenario upon which the parties wish to agree.

However, by treating part of the reusable model as an abstract syntax, to produce a concrete SLA it is now necessary to instantiate concrete classes in the abstract syntax by writing statements in some concrete syntax, such as HUTN. In Figure 3.7, the SLA class is concrete. An HUTN statement could therefore instantiate it as follows:

```
SLA() {

    clientName = "X";
    providerName = "Y";
    serviceName = "Z";
    maxDuration = 10.0
}
```

However, specifying SLAs in this manner raises a difficulty when defining a language for ASP SLAs, as concrete classes must be included in the abstract syntax for every type of statement that the author of an SLA wishes to write. Because a general language is needed to express all SLAs, this implies that an ASP SLA language should have general expressive capabilities. This could be regarded as implying that we should just use UML and OCL to express our models. Alternatively, general modelling language facilities could be embedded in the ASP SLA language.

Instead of following either of these approaches, I have chosen to resolve this issue by deciding that the abstract syntax of an ASP SLA language will need to be "doubly abstract". Not only will it not completely describe the concrete syntax in which an SLA should be written (the usual interpretation of the word 'abstract' in the term 'abstract syntax'), nor can it be expected to completely specify the content of SLAs (the sense of the term 'abstract' applied to classes in object-oriented models). It will rather expresses as much as can be anticipated about the categories of statements that will be needed, using abstract classes and operations.

This means that in order to express many SLAs, such a language specification will first have to be extended, to add concrete abstract-syntax classes capable of expressing the desired SLA. These concrete classes will provide details which previously could not be anticipated, such as the detailed functional behaviour of a service, the nature of any real-world activity being constrained, the scheme by which timeliness, reliability and throughput clauses should be parameterised, or the details of a compensation scheme.

Note that in cases where complex or unusual SLAs are required, this activity will always be unavoidable, and the distinctions between extending a language specification, adding detail to a reusable object-oriented model, or otherwise writing SLA terms in a general-purpose language are irrelevant. Relying on language extension rather than embedding a general-purpose language in the ASP language seems to lead to less redundancy in the overall set of language specifications used, since a meta-modelling language will always need to provide general modelling capabilities if it is to express semantics.

The relegation of extensibility to the definition of the language, rather than the language itself seems to better accord with the observation that the design of SLA conditions can be a complicated matter, in which a highly expressive language is required to produce a statement that must meet several exacting requirements, including those for monitorability, precision and non-exploitability.

Modifications to the language to allow the expression of new types of constraints require extra language facilities that can be delivered in the meta-modelling language but kept separate from the SLA language. They will hopefully be reused in several SLAs, and can therefore be made the responsibility of a language designer, rather than that of individual negotiators of SLAs.

What is therefore important is to structure an abstract SLA language in such a way that, for any new situation, as little extension as possible is required to the core language, and in such a way that it is obvious how extensions should be provided. It is in this sense that I recommend that an ASP SLA language be designed to be *extensible*. The facility of EMOF (and UML) to include abstract classes and operations in a meta-model has the potential to support extensibility in an efficient manner: first, they may be used to provide a framework of abstract classes and operations so that it is clear to the author of a language extension how it should be integrated with the base language – and so that for most extensions it is possible to identify appropriate extension points from which to proceed; second, abstract classes can incorporate concrete elements, in the form of properties and operations, which may implement any anticipated facilities upon which an extension will necessarily depend.

These recommendations may be justified with respect to the requirements for ASP SLA languages as follows:

1. Expressiveness – an ASP SLA language specified according to my recommendations will not be capable of expressing all SLAs meeting my SLA requirements. However, extensions of the core language should be;

2. Understandability – such languages will have a language specification that may be regarded as an object-oriented model of the way that SLA terms parameterise correct behaviour in a service provision scenario. It should therefore be highly understandable;

3. Precision – since such a language will be written in a combination of natural language, EMOF and OCL, it has the potential to be very precise.

4. Restrictiveness – extensions to such a language will produce highly restrictive abstract syntaxes, capable of expressing a small range of good SLAs. The combination of these extensions with standard generic concrete-syntax standards such as XMI and HUTN will result in concrete languages

amenable to a high degree of automated checking.

5. Ease of use – the HUTN standard, and the high automatability of language specifications extended from the core language should make the language easy to use.

6. Power – individual extensions to the core language should have extremely high expressive power, as most of the details of the scenario will be captured by the language specification, with only SLA parameters relegated to a concrete SLA artefact.

7. Automatability – language specifications in EMOF and OCL are intrinsically highly automatable. In particular, the use of the JMI standard allows the automatic generation of meta-data repositories that can form the basis for tools to manipulate language statements;

8. Analysability – the combination of EMOF, OCL and natural language is an extremely expressive language in which to describe a language. This presents formidable reasoning challenges. However, language specifications produced in this manner are no more expressive than the domain models presented in OMG domain standards and profiles. Therefore, the language specifications should benefit from any automated analysis theory developed to support the use of multiple DSLs in an MDA development. This may include consistency maintenance, and the ability to transform models in order to derive analyses [115]. Being machine-readable, such language specifications will be amenable to automated analysis themselves, such as the calculation of metrics as proposed in Section 4.5.

3.2.4 Consequences of the recommendations

The problem with defining a language that requires extension to express all required statements is that it arguably doesn't meet its expressivity requirements. Extensions of the language may be, in effect, different languages. This is very much the case for previous SLA languages such as WSLA [34], WSOL [132] and WSML [112], which rely on extension so much that the core languages provide virtually no practical support for defining SLAs. The authors of these languages enthusiastically promote them as being appropriate frameworks for new languages, a claim which is hard to refute.

This raises the question of how to assess how much benefit an extensible core language for SLAs provides. Furthermore, although a language designed to anticipate extension in this manner is relieved of the burden of providing general expressiveness, which is assumed by the meta-modelling language, the language need not be entirely abstract. An initial version of the language should include all that the support that can be reasonably anticipated for expressing SLAs, given what can be anticipated about the domain. However, over time it may become clear that the same extensions are frequently required. These may be integrated into future versions of the language in an attempt to improve its expressive adequacy. However, this will raise the question of whether the language is genuinely being improved by such additions, or merely being rendered more complicated and harder to use. This question is addressed further in Section 4.5, in which I develop a set of metrics that attempt to measure the usefulness of a language specification, which may be extensible.

3.3 Other approaches to defining languages

3.3.1 Specification of syntax

A good deal of prior research concerns the specification of syntaxes that are more or less concrete. This work can be broadly divided into two categories: that which is primarily concerned with investigating the theoretical properties of syntaxes; and that which is concerned with providing support for engineering language tools, in particular compilers, interpreters, serialisers and deserialisers, and consistency checkers.

The former category is of little relevance to this work. Probably the most commonly used approach to specifying the structure of a language for theoretical purposes is to use a constructive formal grammar, which specifies a language consisting of a (possibly infinite) set of strings [73]. Formal grammars are typically not abstract, as they usually contain terminal symbols from some alphabet. They are not universally practical as the basis for producing tools as efficient algorithms for parsing them do not exist for all classes of formal grammar.

Context-free grammars are a restriction of formal grammars, commonly expressed using a notation called Backus-Naur Form (BNF) [45]. BNF can easily be encoded in a machine-readable form, and is probably the most venerable approach to specifying concrete syntaxes in a manner that is useful for automatically generating language tools. Further restrictions to such grammars enable efficient parsing algorithms to be implemented. These restrictions rely on limiting the amount of lexical context needed to determine what grammatical production is currently being parsed, and tend to result in languages similar in style to most modern programming languages, many of which are defined in this manner.

This approach to defining the syntax of languages is practical, and can be used in conjunction with a traditional style of semantic definition, as described below, to define a language with precise semantics. However, it also has its deficiencies, in that, in comparison to abstract syntaxes, a generative syntax typically only admits of a single representation. This is inconvenient in the case of ASP SLAs where we may wish to have several representations that are more suitable for either human-use (e.g. HUTN), or machine processing (e.g. XMI). Also, this approach would rule-out the use of model-denotational semantics, which require an object-oriented abstract syntax, and which, as discussed below, have several advantages compared to more traditional approaches. The HUTN standard maps an object-oriented abstract syntax to a BNF grammar, so some of the usability properties of a context-free grammar may also be obtained when specifying an abstract syntax.

The other major approach to defining syntaxes for use in practice is that taken to define markup languages, such as the Hyper-Text Markup Language (HTML) [138] and dialects of the eXtensible Markup Language (XML), which derives from an approach originally defined for the Standard Generalised Markup Language (SGML) [35]. These approaches assume that a document conforms to a loose 'concrete reference syntax' which typically subdivides the document into tags, which define a labelled, hierarchical structure for a document. A document so structured is deemed to be well-formed. However, further constraints specific to HTML or XML dialects then restrict the structure and content of tags, resulting in a stricter standard of validity to which documents must conform. Since well-formedness is

assumed, validity rules can be expressed concisely in a special language, resulting in a Document-Type Definition (DTD). DTDs, the structure of which is defined by the SGML standard, are not highly expressive of structural constraints. The XML Schemas specification has been proposed as an alternative for XML dialects [24].

Various versions of the XMI specification provide a mapping from abstract syntaxes to both DTDs and XML schemas, so again, defining a language using an abstract syntax expressed using EMOF can be regarded as equivalent to defining an XML grammar. However, OCL constraints, which may be included in an object-oriented abstract syntax, are more expressive than the constraints that may be included in XML schemas. It is therefore possible to specify a syntax with more precision using a combination of EMOF and OCL than using an XML schema.

3.3.2 Specification of semantics

A number of approaches have been developed to provide a specification for the meaning of languages that is in some sense 'formal' or precise. Although specifying the syntax of a language using an abstract syntax is not radically different from other popular approaches to defining syntaxes, due to the mappings provided by the XMI and HUTN standards, choosing to use a model-denotational approach to specify the semantics of a language requires more justification in comparison to the alternatives.

Informal approaches

The most rigourous approach taken with previous efforts to define SLA languages has been to attach natural language descriptions to syntactical elements in a systematic manner directed by the structure of the syntax (see the survey of SLA languages in Appendix A). Variations on this approach are driven by variations in the style of syntactic definition. WSLA and WSML use XML schemas, and hence document each XML schema type. Other languages may use a syntax expressed in BNF, and so will tend to document the meaning of individual productions in the grammar. Languages such as UML and MOF that rely on object-oriented meta-models attach descriptions to the types and relationships in their meta-models. As discussed in Section 3.1.4, these approaches begin to suffer from ambiguity when the structure of the language is dissimilar from the structure of the domain, as is the case with SLAs.

Classical formal approaches

More formal approaches tend to introduce an intermediate model between the language and the natural-language description of the domain. Classical examples of this are axiomatic, operational, and denotational semantics, all of which introduce an abstract model of a domain independent of the syntax of the language, normally expressed using traditional mathematical logic or set notations. Each approach defines in a different way the effect of statements in the language on elements in the domain, and are typically applied to programming languages.

Axiomatic semantics [123] define the effect of operations in the language by defining a set of statements that are always true about the state of a system and the operations that it can perform. These axioms can be combined with a program specification to derive information concerning the effect of the program on the state of the system. Axiomatic definitions are primarily useful for proving properties of algorithms, and it is not clear how they could be usefully applied to the definition of a language for specifying constraints on behaviours, such as an SLA language.

Operational semantics [106] define the effect of a language by identifying the changes in state effected by an operation, or by sequences of operations. An operational semantic definition has been successfully employed in languages for performance analysis. Two very similar examples are PEPA [33] and TIPP [32], both of which are stochastic process algebras. Statements in these languages define abstractly concurrent processes active in computing systems as sequences of actions having approximate completion rates. Processes can be synchronised on shared action, and may also branch depending on the value of state variables or non-deterministically. Operational semantics define the effect on the current state of the system caused by the completion of actions, effectively defining finite graphs of states, the nodes of which represent combinations of actions contending to complete next.

These languages are highly amenable to analysis, as the state graphs implied by the semantics can be rolled out, completely or heuristically, to detect problems with processes, such as deadlock or livelock waiting for contended resources, and quantitative properties such as average response time and throughput for processes.

Such an approach could conceivably be the basis for the definition of an SLA language, and the benefits in terms of analysability would be considerable. Several problems render this approach undesirable. The semantics of languages such as TIPP and PEPA are approximations of the real behaviour of the system, incorporating assumptions, such as the notion that actions have a constant risk of completion, that are simply untrue of the systems that are being analysed. This degree of approximation is inappropriate for SLAs, the intent of which is to define constraints on actual systems. Removing the assumptions renders analysis computationally infeasible, eliminating the original advantage of the approach. Moreover, although actions, or operations are clearly performed in ASP services, it is not clear that this is the ideal primitive notion for expressing behavioural constraints. Actions are typically performed within the infrastructure of a single party, and hence are not monitorable. Of more interest are the events arising from actions, which may be observable by multiple parties as a result of interactions.

In a denotational approach [123], syntactic elements are defined as being indicative or equivalent to the presence of elements in some domain model, or in the case of operations, to functions transforming the state of a domain model. Although originally developed to describe the behaviour of imperative programming languages, a denotational approach is quite appropriate for more declarative language (as needed for SLAs) due to its ability to describe the way that syntax implies either static or dynamic domain elements. However, a traditional denotational approach typically defines its semantics using a function that maps from statements conforming to a formal grammar, into an abstract mathematical domain, for example a tuple-space. The function and the domain may or may not have an intuitive interpretation, as the principle objective of a denotational approach is often to prove various types of formal equivalence with an operational semantic description, for example that every distinct program in a language produces a distinct result (i.e. has a distinct denotation).

The model-denotational approach can be seen as a straightforward application of the denotational semantics approach, encoded into an object-oriented formalism (the meta-modelling language). It is not traditional in the sense that the domain model is not expressed using a mathematical domain, but instead

using a similarly expressive combination of meta-modelling and logical constraint language. Although less theory exists to support reasoning with model-denotational semantic descriptions than is the case with formal mathematical descriptions, the advantage is that the domain of the language is described in a more understandable manner, and is more immediately suitable as an artifact in software engineering activities. Moreover, the model-denotational approach associates a semantic domain model with an object-oriented abstract syntax, rather than a formal language, conveying the benefits of this approach to defining syntax, as discussed above.

Reusable domain models

The inclusion of a domain model in an ASP SLA language meta-model requires a development effort in similar category to previous work to produce reference models for various types of systems or information. The intent of such efforts is usually to standardise vocabulary or data-models to allow greater inter-operability of development efforts or systems. Two such efforts that are notably similar to our work are the Common Information Model (CIM), and the Reference Model of Open Distributed Processing (RM-ODP).

CIM [18] is a model of management information in a computer system. It is defined in a textual syntax called the Managed-Object Format (MOF), which is similar but distinct from the OMG Meta-Object Facility (MOF) language. CIM's specification consists of the definition of CIM's MOF, plus an extensible model of system management information. This large model includes classes of metric and measurement information, and other information similar to SLAs such as policy goals. It could potentially be extended to include SLA information. The model is intended to be instantiated as a database, in a similar manner to instantiating a JMI repository from an OMG MOF meta-model. The semantics of the elements in the model are hence primarily defined in terms of instantiated data-structures. However, the CIM MOF definitions of elements allow the embedding of human readable documentation that also defines the intent of the element with respect to the representation of a managed system.

The RM-ODP is an extremely influential specification. It defines a collection of five related viewpoint languages for describing distributed systems:

- enterprise viewpoint – focussing on purpose, scope and policies for the system;
- information viewpoint – focussing on information structure and processing;
- computational viewpoint – object-oriented descriptions of systems;
- engineering viewpoint – the relationship of the system with software infrastructure, such as middleware;
- technology viewpoint – the deployment of systems.

The standard can be considered a reference model in itself in so far as it describes the semantics of each of the viewpoint languages with varying degrees of formality. The concepts on which the languages are based could be referred to in the definition of SLA languages. For example, the technology viewpoint can describe the deployment of systems distributed across a network. QoS is defined as a property of

object systems, and related to a notion of contracts that express the expected behaviour of those systems. The Business Contract Language (BCL) [54] bases its definition of a legalistic contract on an extension of the semantics for communities included in the enterprise language.

RM-ODP is defined using a mixture of English-language descriptions and a formal operational semantics specification for the computational language. Reusing the semantics in a model denotational approach could be achieved via the intermediary of the EDOC profile [82] for UML. The purpose of this profile is to extend UML with RM-ODP. However, according to the current fashion for profile documentation, it also defines domain models directly using the MOF language, effectively defining a meta-model for RM-ODP concepts. These models could be augmented with SLA concepts and related to the syntax of an SLA language.

Reusing part or all of the structure and semantics of these models in a reference model for service provision (the semantic part of an ASP SLA language specification) would confer various interoperability advantages, as well as lending the language the authority conferred by these specifications. Using the CIM model would make it easy for monitoring solutions for the language to inter-operate with existing CIM repositories. Indeed, CIMs reliance on extension and its support for instantiating custom information repositories suggest a possible implementation strategy for SLA monitors. Similarly, greater compatibility with the RM-ODP standard would improve the utility of the specification in software development projects relying on that model.

However, the integration of either of these specifications into a language meta-model has drawbacks. The SLA language may become less specific to the expression of SLAs. Favouring one model over the other may result in adoption challenges for the community associated with the model that was omitted. Finally and most importantly, the world view of SLAs typically deals with interactions between parties (as discussed extensively in Chapter 5), which are necessarily largely located at the interface between parties, and is not directly compatible with the views of CIM and RM-ODP, which focus on providing a vocabulary for service infrastructure and provisioning, which are largely contained within the respective administrative domains of individual parties in the ASP scenario. Therefore I have not pursued this approach in this work. However, integration with standard models should not be ruled out in the future, and the recommendations for the development of ASP SLA languages presented here are not incompatible with such an effort. The object-oriented approach taken is compatible with the meta-theories of CIM and the EDOC profile, and with the object-oriented world-view of RM-ODP.

3.4 Summary

In this chapter I have developed a set of three recommendations for the definition of a language for ASP SLAs, as follows.

1. a language for ASP SLAs should be specified using a combination of the technical languages EMOF and OCL, and natural-language descriptions;

2. a language for ASP SLAs should be modelled using the model-denotational approach to provide both an abstract-syntax for the language, and a precise description of the semantics of the language;

3. a language for ASP SLAs should be abstract and extensible to best address the tradeoff required between restrictiveness and expressiveness in the ASP SLA domain.

To support these recommendations, I have described the role of the EMOF and OCL language within the broader context of the standards supporting the OMG's MDA initiative, permitting a justification of these languages as appropriate for the specification of DSLs. The discussion of the MDA also provides the context for contributions discussed in the next chapter.

I have also looked at object-oriented modelling using a combination of class-diagrams and OCL. This serves two purposes: first, as a foundation for the explanation of the model-denotational approach to defining languages, which involves the combination of two object-oriented models defining the syntax and domain of a language; second, it allowed me to argue that object-oriented modelling benefits from good properties of understandability and precision.

I demonstrated that an attempt to develop reusable object-oriented models of SLAs would tend to result in models which resembled language specifications using the model-denotational approach. Therefore, this approach will benefit from the same properties of precision and understandability, and should be preferred over the use of plain object-oriented modelling because it also enables the use of standard concrete syntaxes, such as XMI and HUTN, for expressing SLAs, and introduces a categorical distinction between specifying SLAs and developing new SLA syntax, which may be promote a more responsible approach to producing good SLAs.

However, this approach suffers from a conflict between the expressiveness requirements of ASP SLA languages, which must encode conditions relating to a range of external factors of boundless diversity, and the requirements that a DSL be powerful and restrictive to reduce the cost of producing statements. I argued that ASP SLA languages, if they are not general (harming restrictiveness), must necessarily be extensible, and proposed that abstract classes and operations should be included in the language specification to guide the production of extensions, and to allow the specification to contribute to defining conditions even though the complete details of what is required cannot be known in advance.

Finally, I provided additional arguments in support of these recommendations by considering how following them would influence the degree to which a language would meet the requirements described in Section 2.8, and by comparing the approach favourably to alternative approaches for defining the syntax and semantics of domain-specific languages.

Chapter 4

Domain-specific language specifications

In the previous chapter I have described an approach to defining a DSL for SLAs by modelling the syntax and semantics of the language using a combination of the EMOF and OCL standards. Defining a language according to these recommendations results in both a formal meta-model artifact and a natural language description of a language that ultimately establishes the semantics of the language. It may also result in some other sources of information concerning the language, such as requirements or design documents. Some of these, the designer of the language will consider to be definitive of the language, some will merely be useful for understanding the language, and others may be irrelevant or obsolete. The definitive sources of information concerning the language must somehow be delivered to a user of the language, and confusion must be avoided between definitive and non-definitive artifacts. In this chapter I assume that the definitive sources of information concerning a language can be grouped together into a single artifact that will be made available to the language user, which I call a *language specification*.

In this chapter I present a collection of contributions to the state of the art in defining, utilising and reasoning about language specifications, of the kind that naturally result from following the recommendations presented in the previous chapter.

As discussed in the previous chapter, the approach used by the OMG, the foremost standardiser of languages defined using object-oriented abstract syntaxes, is to publish a PDF document describing both formal and natural-language components of a language description. This has the disadvantage that that the software-engineering benefits of having a formal specification of the language are largely lost. Moreover, these specifications, although definitive of a language, are typically not referenced in an operational context. Therefore the association between artifacts defined in a language specified in this way and the language itself is often unclear, potentially resulting in misinterpretation of the artifacts. In this chapter I first describe some modifications to the MOF standard, and related concrete-syntax standards, that are required to address these issues. These modifications improve the potential of the specification of an SLA language defined using these technologies to meet the language-specification requirements defined in Section 2.9, and SLAs written according to the specifications to meet the precision requirements specified in Section 2.7.3.

Next, I describe the tooling that is made possible by choosing this approach to defining DSLs, which I have implemented in an open-source project called the UCL MDA tools. I describe the use of these kinds of tools to automatically generate a checker component, capable of determining whether a model

of service provision complies with an SLA. Such checkers allow the testing of a language specification, and I also evaluate the performance of a checker employed as part of a runtime monitoring system for SLAs.

Finally, I describe a set of metrics helpful in evaluating the usefulness of DSLs. These metrics are types of measurements made over language specifications and statements in DSLs.

4.1 Language specifications as first-class entities

A concrete SLA is intended to capture and preserve the intent of some parties with respect to an agreement concerning the provision of some service. This raises two potential problems:

First, as described in the previous chapter, a reasonable approach to balancing the conflicting needs for a highly expressive language, able to capture any SLA terms, and a language that reduces the cost of SLA preparation by providing reusable domain knowledge is to provide an abstract, extensible, domain-specific language. However, to make use of such a language, extensions to it will frequently have to be defined. This means that the author of an SLA will need to keep careful track of what extensions they are using, what the extensions mean, and how the extensions combine with the core language.

Language extensions must therefore be understandable and precise, just as a core-language specification must be understandable and precise, so that the author of an SLA does not introduce errors due to a lack of familiarity with a particular extension, or confusion between the meaning of several similar SLA-language extensions, and also so that the precise meaning of a concrete SLA, defined with the help of language extensions, can later be recovered.

Second, the requirement that SLAs be precise also implies a particularly strong need to preserve traceability from the concrete expression of the SLA to the semantics of the language, including the specific extensions being used. As potential components of legal agreements the meaning of an SLA must be as completely and precisely defined as possible. Clearly if an SLA contains no reference to its semantics then the meaning of the concrete document can be disputed after an agreement has been made. Asserting a link between an SLA and the language in which it is defined becomes more difficult if the definition of that language is spread across a standard core-language specification and several non-standard extensions.

The approach to defining an SLA language presented in the previous chapter advocates the use of a combination of EMOF, OCL and natural language to define a language for SLAs. I recommended that in the case of an SLA language it was helpful to use the model-denotational approach to provide precise semantics for the language. I also described the way that these (or similar) languages are used to specify existing OMG languages: MOF is used to define an abstract syntax; OCL to refine the abstract syntax with constraints; these formal definitions are then described in a specification document, typically a PDF, using natural language, usually with the assistance of diagrams; natural language is used to establish the semantics of the language, with or without an intermediate domain model of some kind.

From the need for language extensions to be as understandable and precise as core-language specifications, it follows that these extensions must be documented to the same standards. Adopting the OMG's approach to producing language specifications, it would be necessary to produce at least one

new PDF document for each new extension defined. The relationship between any extension and the core-language that it extends must also be clearly defined. This could documented with the extension, in an additional document, or the combination of the extension and the core language could be treated as a new language and re-documented in a stand-alone language specification. Clearly, any of these choices requires a significant effort in document preparation. Whilst the resulting PDF documents may very well be helpful, they would be expensive to produce manually. Moreover, in contrast to a formal language definition such as a MOF XMI document, these PDF documents cannot be automatically checked for consistency, or used to support the automated reconfiguration of tools for editing and checking SLAs, facilities that will be important if numerous language extensions are being defined.

Conversely, choosing to rely on one of the standard formal sources of information concerning a language is also unsatisfactory. Standard MOF XMI documents do not include the natural language commentary necessary to understand the semantics of a language, and may not include important auxiliary definitions such as OCL constraints.

An additional problem with existing standards is that the concrete-syntax standards compatible with an approach to defining languages using object-oriented abstract syntaxes, such as XMI and HUTN, do not adequately preserve traceability between statements that make use of them, themselves, and the language specifications by which they are parameterised. Therefore, an SLA encoded using the standard XMI approach will not have any explicit link to the XMI standard, or the SLA language specification in which the meaning of the SLA is defined. Hence, it may be possible to contest the meaning of such an SLA, reducing its effectiveness as a means to mitigate risk.

These deficiencies in the OMG's standards and its approach to defining languages also adversely affect languages defined for other purposes. For example, in a software development effort, it is frequently important that a document in some language both adequately captures the author's intent and reliably preserves that intent across time and space. For example, this is true of requirements specifications, models, and even program code. Several people may work in a software development endeavour, and they will communicate not only verbally but through shared artifacts. These artifacts also serve as repositories for knowledge that may not be faithfully preserved in the memories of developers.

It is reasonable to expect that any reader or modifier of a statement in a technical language should be able to understand it in the same terms as its original author. Although in broader contexts it may be appropriate to assume that a subjective component exists in the interpretation of any text, in SLAs, software development, and engineering more generally, this can lead to expensive mistakes. Where ambiguity exists in an artifact, disambiguation may be possible by contacting the author of the artifact, but this is not always true. The author may be unavailable for some reason, or the author and recipient may be the same person, separated only by an interval of time during which the vital information has been forgotten. Therefore, properties of precision and understandability must inhere in the artifact.

A lack of semantic ambiguity is particularly important in the context of an MDA development. Although no strong definition of what constitutes such a development exists, it seems likely that the primary type of artifact in such developments will be models, expressed in languages defined using

OMG meta-modelling technologies. Furthermore, given the dichotomy that exists in most formulations of the MDA approach between concepts that are in some sense generic, or 'platform-independent', and those that are specific to particular concrete implementations of a system, it seems likely that several languages, or language variants will be used in any given development [85]. Because the meaning of software development artifacts is partly determined by the meaning of constructs in the languages in which they are defined, this proliferation of languages introduces several pitfalls for development. The true intent of a developer may not be captured by an artifact because a developer has failed to correctly understand the meaning of a language construct that they have employed, a possibility that increases in likelihood with the number of languages used. A failure to preserve an explicit link between artifact and a full syntactic and semantic definition of the language in which it is written may lead to errors in later interpretations of the artifact. The latter issue is of acute importance in MDA development, described in Section 3.1.2, as it is a principal claim of the approach that systems can be redeployed as hardware and middleware standards change, depending on the reuse of models over an extended period of time.

This section reproduces material first presented in [118] to argue for some relatively minor revisions to existing OMG standards to address these issues.

In Sections 3.1.3 and 3.1.4 I have considered the way in which DSLs are specified in OMG standards, both in terms of their syntax and semantics. These approaches all conform to two observations. First, the definitive reference for a language is the specification document, not a technical artifact such as an XMI encoding of a meta-model. Second, the semantics of the languages are always ultimately defined, directly or indirectly, by natural language statements. Remove these statements from any formal description of a language, and the description is likely to become impenetrable and useless, regardless of the sophistication of any intermediate models employed.

In the next section I consider various schemes by which definitions of syntactic elements are associated with instances of that syntax.

Based on these analyses I propose that all meta-models should embed some definitive documentation that is at least expressed in natural language, or some form that is ultimately documented in natural language, and encodings of these meta-models in a concrete syntax should replace PDF documents as the definitive artifact for a language. I call such documented meta-models *language specifications* because in practice they resemble the specification documents published by the OMG, but unlike the OMG's specifications it is straightforward to retrieve the structure of the language, its constraints and semantic documentation automatically. Making use of the packaging mechanism provided by MOF, it is easy to incorporate the definition of extension elements into copies of these documents, resulting in new, combined language specifications that can serve as a definitive point of reference for concrete artifacts using the extended language. I describe in detail the contribution of this prescription to addressing the twin problems of capturing and retrieving developer intention.

4.1.1 Referencing languages from models

Arguably, only three standards for concrete syntax for MOF-defined languages are published by the OMG. These are the XML Metadata Interchange (XMI) standard [94], the Human-Usable Textual

Notation (HUTN) standard [91], and the diagrammatic concrete notation recommendations included in the UML superstructure specification [81]. Also of note is the UML Diagram interchange standard [95], which allows the encoding of UML diagrammatic notation as XMI or Scalable Vector Graphics (SVG) [141]. However, it does not introduce any new concrete syntax. Standard profiles also render legitimate the use of certain strings for stereotypes and tagged-values in UML diagrams. However, the syntactic rules governing such extensions are fully defined in the UML superstructure specification.

In this section I consider the degree to which concrete artifacts expressed in these syntaxes reference the syntax and semantics of the language in which they are written.

The XMI standard has undergone a number of revisions, and implementations are in use according to several of these. XMI standards map MOF models to XML grammars. In XMI 1.2 a mapping to an XML DTD is defined [83]. According to standard XML syntax, this DTD may be referenced in the header of any document, thereby identifying the syntax of the language in which the document is written. Furthermore, the XMI 1.2 standard acknowledges that DTD syntax is not as expressive of syntactic constraints as MOF models. Therefore an XMI specific header element may be included that includes an optional reference to an XMI file containing the MOF model for the language, providing a better reference for the expected syntax.

In this scheme, interpretation of the link to the abstract-syntax of the language relies on the semantics of the document header, as prescribed by the XMI standard, being understood. However, no unambiguous reference is made to the XMI standard, so an interpreter is not guaranteed to be able to identify a sound basis for interpreting the document. This may hinder the identification and interpretation of the meta-model XMI as a reference for the syntax of the document.

Another important inadequacy in the XMI 1.2 language referencing scheme is that the semantics of the meta-model are not referred to, either from the instance document, or from the referenced XMI for the meta-model. In fact, in the cases of both UML and MOF, the referenced meta-model cannot even be considered to be a definitive statement of the abstract syntax of the language being used, as this is finally established by natural language statements in the PDF standards document for the language.

XMI versions 2.0 [93] and 2.1 [94] revise the standard to generate XML schema, rather than DTDs. Schema specifications of the grammar of an XML file are more expressive than DTDs, and can therefore capture more of the constraints inherent in the source meta-models. Perhaps as a result, the ability to reference the meta-model XMI has been dropped from the standard. This is deeply to be regretted, as no link is preserved to the true syntax of the language. Also, a clue as to the possible applicable language specification has been removed.

XMI version 1.2 and 2.0 depend on version 1.4 of the MOF standard [80]. Version 2.1 of the XMI standard depends on version 2.0 of the MOF standard [79]. Both versions of the MOF standard allow the embedding of syntactic constraints in an arbitrary constraint language. A common constraint language employed is OCL, which is more expressive than XML schema. In no version of XMI are these constraints mapped into either DTD or XMI schema. Therefore XMI version 2.0 and 2.1 are seriously negligent in not referencing the meta-model definition, as some syntactic constraints will be entirely

inaccessible.

The HUTN standard also maps MOF models to a grammar, in this case specified using BNF. The mapping from meta-model to grammar is customisable using a 'configuration', an instance of a configuration meta-model. Details of the configuration used can be included in a discriminated comment at the start of a document in a HUTN syntax. Although configurations can reference elements in a meta-model using their fully-qualified MOF names, nothing in the configuration or elsewhere in a HUTN document references the location of a concrete artifact defining the meta-model for the language used. Moreover, no reference need be made to the HUTN standard, and the inclusion of configuration information is optional, so confusion can potentially arise regarding the syntax to which the document is intended to conform. Finally, no reference need be made to any semantic specification of the language in the document.

UML diagrams are an extremely well-known notation. However, they are potentially highly ambiguous artifacts. No reference need be made to the UML standard in diagrams conforming to any revision of the concrete syntax standard. Moreover, since diagrams only ever display projections of a model, a diagram in isolation is frequently not enough information to determine the true intent of the author. For example, attributes, associations and classes may all be suppressed in class diagrams. To eliminate this ambiguity, diagrams should at minimum refer to a concrete representation of the model that they represent. This concrete instance could then identify information about the language being employed, such as the revision of UML being used, and any profiles being employed.

The concrete instance of a UML model will also contain any profile packages, enabling the use of particular stereotypes and tagged values. Profile packages, stereotypes and tagged-values have no properties referring to any definitive statement of their semantics.

4.1.2 Suggested revisions to OMG standards

To interpret a concrete artifact, it is necessary to understand its concrete and abstract syntaxes, and its semantics. Faced with a concrete artifact, a human interpreter should not have to guess what concrete syntax is being employed, in order to establish a basis for an initial interpretation of the artifact. Therefore all concrete artifacts should include a human-readable comment referencing the specification of their concrete syntax.

Generic concrete syntax standards, applicable to multiple abstract syntaxes, can be adequately defined in current human-readable standards documents. Specialised concrete syntaxes may be documented with other aspects of the language to which they apply. In either case, having established the concrete syntax of an artifact, the abstract syntax and semantics of the artifact must then be identified to obtain a complete interpretation. The concrete syntax standard should permit the artifact to be interpreted sufficiently that unambiguous links to specifications of the abstract syntax and semantics can be followed if they are separate from the concrete syntax standard.

Clearly there are a number of different possible schemes whereby abstract syntax and semantics can be referenced. The meta-model of the language and the specification document for the language could both be referenced. The meta-model could be referenced, and contain a reference to the specifi-

cation, or vice-versa. Alternatively, because the specification documents both syntax and semantics, the specification alone could be referenced.

I prefer the final approach because a language specification provides the definitive documentation for a language. Hence I argue that language specifications, not meta-models should be regarded as first-class entities in MDA developments.

Abandoning the use of meta-models as a language definition is potentially problematic, because existing language specifications cannot be machine interpreted in a useful way. For example, the descriptions of the abstract syntax included in the UML specification cannot be used to populate the meta-layer of a JMI repository without first undergoing manual translation into a more formal representation such as XMI. I address this issue by proposing that the use of MOF and natural language be inverted. Instead of describing a MOF model using natural language in a specification, a specification should consist of a concrete representation of a MOF model with natural language descriptions embedded, to provide an informal commentary on the aspects of the language that are defined technically, and to define the semantics where a technical language is inadequate. In this respect I propose that the MOF specification be revised to define a language which is somewhat similar to a programming language used for literate programming, in which technical and human-readable aspects provide mutual support [46].

Following this approach also addresses the issues arising from the use of multiple language extensions. Because these meta-models benefit from the packaging mechanisms included in the meta-modelling language (MOF), it is straightforward to combine extension elements with a core-language specification in a new joint model, which can then be referenced as the definitive source of meaning from concrete statements expressed in this extended language.

My proposals are captured by the following particular revisions to existing OMG standards:

1. The MOF 2 standard already allows the inclusion of comments in meta-models. The Comment class should be extended with an attribute that indicates whether the comment is intended to be definitive of the semantics of the associated model element, or is merely an informal remark. A constraint should be added to the specification that all types, associations and references should be associated with a non-empty definitive comment that defines the semantics of the element in a manner that can ultimately be understood by a human interpreter.

2. The MOF standard should be redistributed as an XMI concrete specification, with documentation included.

3. Future revisions of all OMG standards that rely on MOF meta-models should have a definitive form published as a concrete specification, for example in XMI form, rather than the definitive version being a PDF.

4. The XMI standard should be revised to incorporate a reference to the specification of the language used. All XMI files should be required to include an XML comment in natural language identifying the location of the XMI specification so that this link can be interpreted.

5. The HUTN standard should be revised, making it mandatory to include syntax configuration information in instance documents if a syntax configuration is being used, and also making it mandatory to refer to the specification of the language being used. All HUTN documents should be required to include a comment in natural language identifying the location of the HUTN standard so that this link can be interpreted.

6. Future concrete-syntax standards should respect the principle that instances should refer both to the concrete-syntax standard, and to the concrete specification of the language being used (if separate from the concrete-syntax standard as is the case with XMI and HUTN). The reference to the concrete syntax standard must be human readable.

7. Diagrams should unambiguously reference a concrete representation of the complete model that they depict.

8. Profile packages in UML 2 should unambiguously reference a specification of the semantics of the extensions they contain.

The contents of definitive comments associated with elements should permit the interpretation of the element by a user. They may do this by introducing some intermediate formalism, encoded in the comment text in some manner. It may also unambiguously reference external documentation. However, the semantics should ultimately be interpretable by a human. Note that it is hard to imagine a circumstance under which interpretability by a human is not a requirement for a language. A language may be essentially descriptive, and therefore aimed at human interpretation. However, even if this is not the case, humans will typically need to build tool support for interpreting the language (or an alternative language, in terms of which the semantics of the first language are specified). Humans must also either write statements in, or define mappings to, the language.

I do not prescribe any particular scheme by which unambiguous references should be made between concrete artifacts. However, URIs would be an appropriate choice [38].

4.1.3 Consequences of the proposed revisions

In practice these modifications would be relatively painless to implement. Existing specifications could be revised into conforming concrete specifications by editing the existing specifications into comments in an XMI file of the meta-model, simultaneously identifying those parts of the documentation that are definitive, and those which are informal. XMI concrete artifacts can be commented using XML comments to identify the XMI concrete syntax specification.

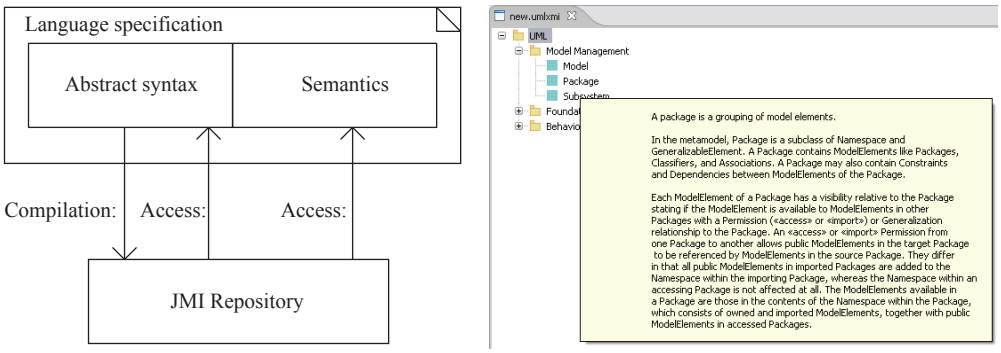
Existing language specifications, such as UML and MOF, include OCL and diagrams in addition to natural language statements and descriptions of meta-models. It is desirable to include this information in machine-readable specifications. OCL statements form part of the definitive specification of the abstract syntax. They also have a semantic character, in that they rule out instantiations of the meta-model that would be illogical given the language semantics. If a model-denotational approach is taken, OCL constraints may also contribute to the semantic definition. Diagrams are typically informal and assist with the explanation of the language and its semantics.

Three approaches can be taken to including this information in concrete specifications. The OCL and diagrams could be included in documentation elements in an unstructured manner. Existing mechanisms for including this information could be employed: CMOF supports the inclusion of constraints with a constraint meta-element; XMI 2.1 supports the inclusion of any kind of auxiliary information using an extension element, or XML namespaces. Finally, the information could be incorporated by extending the MOF specification with the OCL and diagram interchange specifications. Future work will consider trade-offs between these approaches. In terms of the standards I make no recommendation as to which of these approach would be most suitable. However, I took the latter approach to integrate OCL and EMOF in the UCL MDA tools (described in the next section), as it avoids the need for continual re-parsing of OCL embedded in a meta-model.

The need to document MOF version 2 according to its own standards assumes a new significance in this scheme. For a language specification to be understood the meta-language used to document it must also be understood; in the case of MOF this is MOF. The revised MOF specification will be machine readable, and recursively defined, which might seem to make it harder to understand. However, the XMI specification will still be human readable. On this basis, a structural interpretation of the specification can be obtained. The embedded comments in the MOF specification will therefore be extractable, enabling the full semantics of the language to be understood.

These proposals address the issue of capturing developer intent by ensuring that in any context in which the meta-model would otherwise have been employed, for example to populate the meta-layer model in a JMI repository, the semantic definition of the language is available. Additionally the availability of OCL and diagram types ensures that the semantic documentation is available in as machine-readable a form as possible, maximising the potential for using this information intelligently in tools. As a first measure, presenting this information to developers in the same context in which they are using a novel language should assist in ensuring that they use the language correctly. The use of a specification in this manner is illustrated in Figure 4.1. In illustration (a) the relationship between a JMI repository and a language specification is shown. The specification parameterises the generation of the repository interfaces (and potentially implementation). The specification is then loaded when the repository is created to populate the meta-model of the repository, enabling the reflective capabilities mandated by the JMI specification. Illustration (b) shows these capabilities used to good effect in an Eclipse editor plug-in for UML version 1.5 generated by the UCL MDA tools. The tree component is a simple model editor. It has a generic implementation that relies on JMI reflection. Here it has reflectively retrieved the documentation for the `Package` type, which according to my recommendations is embedded in the language specification loaded by the underlying repository. The documentation is presented in a tool-tip, triggered by hovering the mouse over the tree element representing the `Package` class.

The proposals address the issue of recovering developer intent from an artifact by ensuring that artifacts always refer to both any relevant concrete syntax standard employed, and to the concrete specification of the language being used. The concrete syntax is referenced in as unambiguous a manner as possible, using natural language statements included in the artifact. Understanding the concrete syntax



(a) (b)

Figure 4.1: A specification used as input to a JMI generator. Abstract syntax and semantic documentation are available to the repository user via reflection

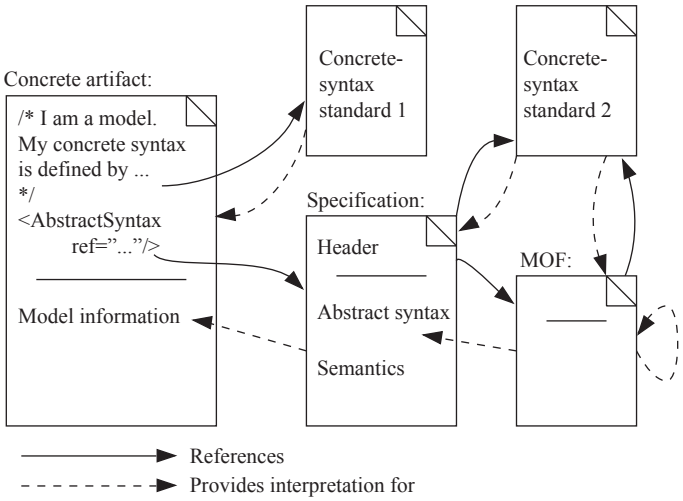


Figure 4.2: Recovering the meaning of an artifact by navigating links to concrete-syntax standards and language specifications

permits the recovery of the concrete specification of the language, which is totally definitive of the language. The concrete specification is constructed on similar lines to the original artifact, so can ultimately be understood in the context of a higher-level language, or in the case of MOF, recursively in terms of itself, and whatever concrete syntax standard is being used to describe it. Figure 4.2 illustrates the process of interpreting a concrete artifact. References in the concrete artifact to documentation for the concrete syntax, and the language specification, provide a basis for interpretation of the document, traversing up meta-layers until a well-known standard such as MOF is encountered.

4.2 The UCL MDA tools

The UCL MDA tools [135] are an open-source project implementing the tool-support on which the evaluation of my thesis depends. The project currently provides:

- a parser and type-checker for a textual concrete syntax of EMOF, in which invariants and side-effect-free operations may be specified using OCL 2; the parser outputs XMI for a conjunction of the EMOF and OCL 2 meta-models;

- A JMI repository generator, taking as input the XMI output of the EMOFOCL parser and capable of producing:
  - standard JMI interfaces for the EMOF meta-model encoded in the input;
  - Java-classes implementing the JMI interfaces to provide an in-memory repository – the implementation uses Java dynamic class-loading to simplify the overriding of the default implementation for specific instance, class-proxy or package-proxy types;
  - standard XMI writers and readers for the language defined by the EMOF meta-model encoded in the input, integrated with the JMI repository to provide serialisation and deserialisation facilities;
  - an editor plug-in, integrating the repository with the Eclipse IDE, and allowing the editing of repository contents using the generic SWT tree-editor;
  - a stand-alone repository editor implemented in Swing, allowing the editing of repository contents using the Swing tree-editor;
  - extended JMI interfaces implementing the listener and visitor patterns, simplifying the implementation of applications that need to track changes in, or traverse data in a JMI repository.
- a generic tree-editor component implemented in Swing, allowing the modification of the contents of repositories generated by the UCL MDA tools;
- a generic tree-editor component implemented in SWT, allowing the modification of the contents of repositories generated by the UCL MDA tools;
- a generic HUTN reader, allowing the population of any standard JMI repository from a HUTN document conforming to the meta-model used to generate the repository;
- an OCL 2 interpreter, including an implementation of the OCL 2 standard library, that can be combined with JMI repositories generated using the UCL MDA tools to evaluate invariants and side-effect-free operations embedded in language specification, or arbitrary expressions parsed at runtime;
- a translator from the output of the EMOFOCL parser to HTML formatted documentation;
- a translator from the output of the EMOFOCL parser to LaTeX formatted documentation;
- a language specification for UML version 1.5, translated from the PDF specification.

The EMOF parser relies on the EMOF meta-model from the draft version 2.0 core proposal of the MOF standard, shown in Figure 3.5, pg. 61. These tools implement the recommendations provided in the preceding section pertaining to the relevant standards, with the exception of explicitly identifying definitive comments, as this is not compatible with this version of the EMOF meta-model.

The textual syntax for EMOF taken as an input to the EMOFOCL parser is similar to Java class declarations, and also somewhat similar to an equivalent HUTN specification. The syntax embeds definitive

comments in a special syntax similar to JavaDoc comments for Java, which leads them to be associated with elements in the resulting model.

The EMOF and OCL meta-models are combined in the meta-modelling tool. OCL constraints can hence be embedded in a language specification also. The whole specification can be parsed to an XMI representation (including XMI for the OCL constraints, and the embedded comments). Syntax checking is performed according to the OCL 2 specification, and my own syntax for textual EMOF specifications. Type checking of all elements is performed according to the semantics of EMOF and OCL 2.

One application of the specification is to reformat it into a format more suitable for human comprehension. I have hence developed a tool, similar to the JavaDoc tool that generates a webpage from an EMOF/OCL/English specification of a language. The structure of the page resembles the structure of an OMG language specification, with each element in the meta-model presented along with its documentation. The page also benefits from hyper-links that cross-reference related elements.

There follows a short example of the tools in use, using the UML version 1.5 language specification. Here is the definition of the UML meta-model class `Class` included in the specification in my textual syntax for EMOFOCL models:

```
/[
A class is a description of a set of objects that share the same
attributes, operations, methods, relationships, and semantics. A
class may use a set of interfaces to specify collections of
operations it provides to its environment.

...
]/
class "Class" extends Classifier {

    /[
    Specifies whether an Object of the Class maintains its own
    thread of control...
    ]/
    isActive : ::Foundation::"Data Types"::Boolean

    /[
    [1] If a Class is concrete, all the Operations of the Class
    should have a realizing Method in the full descriptor. (Corrected)
    ]/
    invariant {

        not self.isAbstract
        implies
        self.allOperations()->forAll(op |

            self.allMethods()->exists(m |

                m.specification = op
            )
        )
    }

    /[
    [2] A Class can only contain Classes, Associations,
```

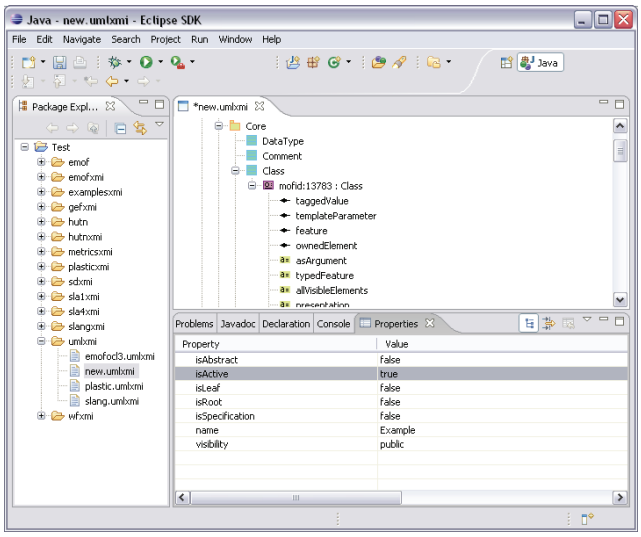


Figure 4.3: Editing a UML class in the Eclipse editor plug-in generated by the UCL MDA tools

```
    Generalizations, UseCases, Constraints, Dependencies,
    Collaborations, "Data Types", and Interfaces as a Namespace.
    (Corrected)
    ]/
    invariant {

        ...
    }
}
```

Note that the definition of `Class` contains the documentation from the semantics section of the UML version 1.5 standard (abbreviated here). The meta-class itself defines one attribute `isActive` of boolean type, in addition to those inherited from `Classifier`, and two additional invariants, of which the second has been omitted here for brevity.

This class declaration is parsed and type-checked by the EMOFOCL parser, resulting in the following fragment of EMOFOCL XMI (from which the comment has been omitted):

```
<EMOFOCL:Class comment="..." isAbstract="false" name="Class"
owningPackage="mofid:2457" xmi.id="mofid:1422">
<EMOFOCL:Class.invariant>
<EMOFOCL:OclExpression xmi.idref="mofid:8171"/>
<EMOFOCL:OclExpression xmi.idref="mofid:12465"/>
</EMOFOCL:Class.invariant>
<EMOFOCL:Class.superClass>
<EMOFOCL:Class xmi.idref="mofid:1427"/>
</EMOFOCL:Class.superClass>
<EMOFOCL:Class.ownedAttribute>
<EMOFOCL:Property xmi.idref="mofid:2799"/>
</EMOFOCL:Class.ownedAttribute>
</EMOFOCL:Class>
```

The EMOFOCL XMI for the language as a whole is then used as the input to the JMI generator, which also generates XMI readers and writers, and the implementation for an Eclipse editor plug-in. Figure 4.3 shows the editor being used to specify an active class called `Example`. When saved, this

model results in the following UML XMI document:

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmlns:UML=
"file:///C:/workspace.jws.3.2.1/UCLUML/gen/uk/ac/ucl/cs/
uml/specification/uml.emofxmi" xmi.version="1.2">
<!--This document is in XMI format according to the OMG XML
Metadata Interchange (XMI) Specification v.1.2, OMG Document
formal/02-01-01 (http://www.omg.org/)-->
<XMI.header>
<XMI.metamodel href="file:///C:/workspace.jws.3.2.1/UCLUML/
gen/uk/ac/ucl/cs/uml/specification/uml.emofxmi"/>
</XMI.header>
<XMI.content>
<UML:Class isAbstract="false" isActive="true" isLeaf="false"
isRoot="false" isSpecification="false" name="Example"
visibility="public" xmi.id="mofid:13783"/>
</XMI.content>
</XMI>
```

Note the XMI header, which implements my recommendation to include a reference to the concrete-syntax standard used, and because XMI is parameterised by a language specification, also the language specification.

This model could equally have been specified with the following HUTN document:

```
// This document is encoded according to the HUTN version 1.0
// specification OMG document formal/04-08-01
// (http://www.omg.org)

specification =
    "file:///C:/workspace.jws.3.2.1/UCLUML/gen/uk/ac/ucl/cs/
uml/specification/uml.emofxmi"

::Foundation::Core::Class() {

    name = "Example";
    visibility = public;
    isActive = true
}
```

4.2.1 Alternative MDA tool support

A number of JMI repository generators have been developed with varying degrees of flexibility in terms of the input format they require and the type of code that they can generate. However, I found none to be ideal for my purposes, and elected to produce my own implementation of the standard.

One of the earliest available JMI generators was that produced by Novosoft [76]. The implementation code that the Novosoft JMI generator produces is hard-coded into the implementation of the generator. Probably the most commercially significant generator is the Eclipse Modelling Framework (EMF) [21]. The EMF generates specific repositories from meta-models according to a pattern similar to JMI. However, it is not template driven, so offers no control over the implementation of the repository. It is also not standards compliant, expecting languages to be defined according to a model called ECORE, which is similarly expressive to EMOF, and producing interfaces that do not comply with the JMI standard.

Another alternative is the AndroMDA tool [2], implemented using Velocity templates. Custom templates can be configured by the user, and the tool parses XMI representations of models and makes available standard context objects. However, Velocity templates do not have powerful control structures. Without the ability to modify the structure of the context objects to preprocess model information it is impossible to generate some desirable outputs using AndroMDA. For example, generating an XML DTD for an XMI reader requires the use of transitive closure across inheritance relationships in the model, which cannot be achieved in the template.

A powerful alternative is that implemented in the Kent Modelling Framework, version 3 [44] (KMF). This tool evaluates string-typed OCL expression over models to generate program text. This approach is potentially very powerful, since OCL is recursive so can calculate arbitrary functions of the model. However, the OCL expressions are hard to write, particularly when a 'generation state' has to be maintained, containing things like a list of unique identifiers used.

The need to maintain the flexibility to change the implementation code generated by a JMI repository generator is an important requirement for these tools. It is generally assumed that repositories will be contained wholly in memory. However, as discussed in Section 4.4, there is likely to be the need in future to implement JMI repositories that are backed by a database to support storage of large models. Moreover, flexibility in the implementation code allows the implementation of non-standard functionality in the repository, such as support for the listener and visitor patterns. The JMI generator in the UCL MDA tools specifies its outputs using a simple template format similar to Java Server-Pages [130]. Embedding of Java code in these templates allows information to be extracted from a JMI repository generated for the EMOF meta-model. The templates are rewritten as Java classes using a process of regular-expression replacements, and when compiled are integrated into the JMI generator.

In early work with these modelling technologies I relied on the KMF for OCL evaluation. Unfortunately I found the implementation to be defective, and it also relied on closed-source libraries. I therefore later implemented my own OCL compiler and interpreter in the UCL MDA tools. However, the design of my interpreter, which uses the visitor pattern to traverse the abstract syntax of the OCL language is very similar to that of the KMF OCL interpreter.

4.3 Testing language specifications

According to my proposals, language specifications may contain both definitive and non-definitive documentation. They may also contain both repository types that are part of the abstract syntax of the language being specified, and other types for the purpose of semantic exposition.

Non-definitive comments included in a specification remark on aspects of the specification already inherent in some definitive part. For example, a human-readable explanation is often given for OCL constraints included in a specification. However, the meaning of the OCL constraint is completely defined by the OCL standard, so this explanation does not refine the specification in any way.

However, non-definitive elements should not be misleading. Ideally the non-definitive parts of a specification should be entirely consistent with the definitive parts. Although such consistency will be extremely difficult to prove conclusively, a language developer may wish to develop confidence that the

correspondence holds by testing the language.

Testing is clearly straightforwardly enabled for language specifications defined according to my recommendations. As discussed above, by defining a language using EMOF, it is possible to generate a JMI repository capable of storing statements in the language. This in itself provides a testing mechanism, as it makes it possible to check that a number of desired statements can in fact be constructed according to the abstract syntax of the language.

Testing of semantic elements in specifications is also possible. Although these would not normally be included in a repository, this constraint can be relaxed to allow these elements to be represented explicitly in the repository.

As well as testing structural properties of the language defined by the EMOF elements in the language specification, it is also desirable to test that the OCL constraints used to refine the syntactic and domain models, and to define the semantics of the language, correspond with the intent of the language designer. This can be achieved by combining an OCL interpreter with the repository, as implemented by the UML MDA tools. Test cases can then be devised to check for under- and over-constraining invariants, and to determine that the results of evaluating side-effect-free operations are as expected.

The possibility to test not only the syntactic but semantic properties of a DSL is a significant advantage of my approach to defining language specifications, and comes without requiring any restriction on the domains of the languages which can be defined (such as only allowing languages with operational semantics that can be executed by a computer).

Checking of this kind has not been scrupulously performed in prior OMG standards. In [9] for example, a significant number of syntactic and type errors are discovered in the OCL constraints included in the latest version of UML, as a result of their formulation without the aid of a parser. It seems highly likely that OCL definitions also exist in the specification that do not perform as expected, or as documented in the informal comments accompanying the constraints in the language specification. An investigation into the semantic correctness of constraints in the UML specification would be productive future research.

This type of testing is of course highly useful for an SLA language, where the parties will wish to be sure that a concrete SLA captures the true intent of their agreement. Since the meaning of the SLA is in part defined by the language in which it is specified, testing the language will increase the confidence the parties have that this is so.

A language tested in this manner from the outset will benefit from an additional source of information concerning itself: the test cases. These will constitute an additional resource for understanding the intent and meaning of the language. They may also be of use in developing conformance tests for automated tools that rely on the semantics of the language.

4.4 Runtime monitoring of ASP SLAs

Parties engaging in ASP SLAs may wish to monitor electronic services. The client of an SLA will wish to know whether the SLA is being violated, in order to take action to claim compensation. The provider will wish to know whether the client is violating the SLA, for example by violating a throughput constraint,

and also if the service is performing in such a way that there is a danger that the SLA will be violated, enabling the provider to take remedial action.

In the previous section, I discussed the fact that a language specification implemented according to the recommendations included in Section 3.2.3 could be tested by generating a JMI repository from the specification, loading that repository with objects representing the syntax of an SLA and also with objects representing the real-world entities and events with respect to which the SLA is defined, and then evaluating some or all of the OCL constraints included in the language specification to determine whether or not the situation as modelled conforms to the SLA. Moreover, the results of side-effect-free operations defined in the specification can be tested similarly.

Of course, once a language specification is tolerably free from errors, the results of such checking can be used to assess whether a given model of a service scenario conforms to the terms of an SLA.

Runtime requirements-monitoring systems typically consist of a set of software instruments for gathering the raw event data pertinent to the properties of interest, some logic for checking that this data meets requirements, and possibly a repository for data if requirements checking needs data gathered over an extended period. Clearly, a JMI repository generated from an SLA language specification has the potential to implement the requirements-checking logic and repository parts of such a system. In [116] and [117] I proposed and investigated the practicalities of taking this approach. This work relied on an early version of the SLAng language, the most recent version of which is described in Chapter 6. The version used benefitted from a abstract syntax supporting latency and throughput conditions, with model-denotational semantics that had been tested to some extent to generate confidence in their correctness. The version differed from the latest version of SLAng in that it was not abstract or extensible. I summarise the findings of this work here.

4.4.1 Architecture of the SLA checker

The SLA checker used in these investigations consisted of three major components:

1. The automatically generated JMI interfaces and implementation for holding SLAs and event data.

2. The Kent OCL implementation, with SLAng constraints loaded, for checking whether SLAs have been violated.

3. An API wrapper, that allows checks to be requested, and returns lists of violations that have been found. This part is hand-written in our implementation, because it is independent of the structure and semantics of the SLAng language.

These investigations predated the implementation of an OCL interpreter in the UCL MDA tools, and hence used the Kent implementation of OCL, which, as discussed above, I later abandoned for reasons of maintainability. The JMI generator was an early version of that now included in the UCL MDA tools.

The checker may be incorporated in electronic service systems wherever SLAs need to be monitored. It is used as follows:

1. The checker is instantiated.

2. The static elements from the semantic model are instantiated or loaded from an XMI file. These elements, with types such as `ElectronicService`, `ServiceClient` and `Operation`, represent knowledge that the checker has about the service or services being monitored. The model is manipulated using the generated JMI interfaces.

3. One or more SLAs are instantiated or loaded from an XMI file, again using the JMI interfaces.

4. Associations are established between the service components defined in the SLAs and those components in the service model created in Step 2.

5. Monitoring data is provided to the component by invoking the various 'create' methods found on the JMI API (e.g. `createServiceUsage()` on the `ServiceUsage` class proxy interface). These data are associated with the relevant static elements in the service model, created in Step 2.

6. Periodically, the `check` methods on the violations API may be invoked. These return lists of violations, if any exist.

The instruments measuring the performance of the service are not part of the SLA checker, so must be implemented separately. For a given SLA, a combination of the descriptions included in its terms section, and the reference model of the service included in the language definition provide the guidance as to what data these instruments must provide.

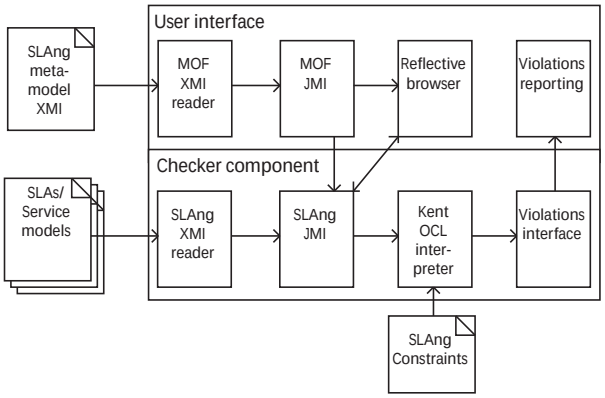


Figure 4.4: Design of the SLA checker

Deployment of the checker component

I tested the SLA checker by deploying it to monitor the performance of an EJB application. The application is an auction management system developed by an industrial collaborator. The application is deployed in the popular application server `JBoss`, which implements the Java 2 Enterprise Edition (J2EE) specification [127], using `Apache Tomcat` to serve the web front-end [5].

The architecture of `JBoss` is based on the Java Management eXtensions library (JMX). In this component-based architecture, all functionality is deployed as 'managed beans' (MBeans), Java components that expose meta-data, configurable properties and lifecycle management methods. The `JBoss` distribution and default configuration includes MBeans implementing EJB containers, JNDI naming services, transactions, and many other services. The SLA checker is deployed as an MBean, meaning that it

has one instance per instance of the JBoss server. It is made available to other MBeans and to deployed EJBs via the JNDI naming repository.

To provide external access to the SLA checker, I implemented a small J2EE application called 'The SLAng Control Panel'. This consists of a single JSP page providing an interface to a stateless session bean. This bean in turn delegates operations to the SLAng checker. The main operation provided by the checker over this interface is checkAll(), which causes the component to evaluate the SLAng constraints over its internal model of SLAs and service data, and return a list of violations, if any exist.

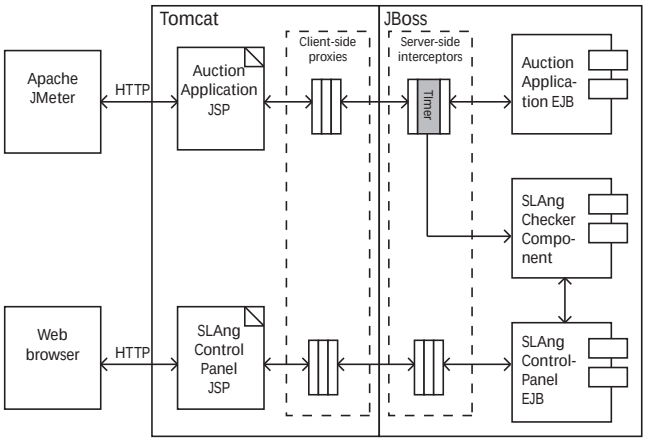


Figure 4.5: The SLA checker component deployed to monitor an EJB application

Service performance information is passed to the SLAng service by a server-side interceptor configured as an option of the JBoss container configuration. Interactions with EJBs hosted within JBoss are implemented using a stack of interceptors on both the client and server side. These allow different types of functionality to be added to the communication channel independently, such as transaction management, security, and the communication protocol itself, which is managed by the outermost interceptor on client and server sides. For the purposes of evaluating the SLAng component, I added an interceptor on the server side to measure time spent processing EJB requests. The interceptor accesses the SLAng service using JNDI and invokes the createServiceUsage(), method on its JMI interface to record the measured time. Apache JMeter was used to generate a variety of loads on the service [6].

4.4.2 Evaluation of the checker component

The SLA checker was evaluated on three points: The ease of implementation of the checker; the ease of deployment of the checker in its intended context (in this case to monitor the auction application); and the performance of the checker.

Implementation

Effort in implementing the checker falls into three categories: implementing the JMI generator; implementing the SLAng language specification that is the input to the generator; and implementing the remaining code for the component, which mainly involves the integration of the OCL evaluator component and the provision of an API for requesting checks and reporting violations. Of these three categories, the first two can be discounted on the grounds that they are separate efforts from the implementation of the actual checker component. Taking this into account, the implementation of the component took around

1 man-week of labour. The SLA checker consists of approximately $115,000$ lines of code (including blank lines and comments) outside of standard libraries, of which $77,000$ were generated, $36,500$ form the implementation of the OCL evaluator and $1,500$ were hand written.

Deployment

The checker was straightforward to deploy into the `JBoss` application server. This is mainly because `JBoss`'s architecture is expressly designed to support the deployment of new services and components. However, the JMI interfaces also contribute by providing a clear API through which to deliver service performance data, and the XMI reader interface and implementation makes loading SLAs and service models into the component simple. Implementing the SLAng control panel application and integrating the component into `JBoss` took 2 weeks for a programmer not previously intimate with the workings of `JBoss`.

Performance

The major problem with the SLA checker is its inability to scale with the amount of monitoring data provided. This is manifest in two ways: first, and most seriously, the time taken to evaluate the OCL constraints is highly dependent on the amount of monitoring data provided, and is far too long for models containing realistic amounts of service data. For example, for a data set of 1000 service usages, the evaluation of a throughput constraint takes 20 minutes on a PC with a 1.7GHz Intel Pentium 4 processor.

The second issue is that the checker has an unrealistically small capacity for monitoring data. In the implementation of the JMI interfaces on which this experiment was performed all data is represented as Java objects stored in main memory. Since no policy for removing or persisting old data was implemented, this leads inevitably to memory exhaustion as the application continues to be used. The amount of service usage data that can be checked is restricted by the amount of main memory available to the virtual machine in which the component is deployed.

Both of these issues represent theoretical challenges that need to be overcome if this approach is to be used in practice.

The first issue is perhaps the most profound. Evaluation of OCL constraints in the version of SLAng used in this test may have been slow for a number of reasons:

1. interpretation of a language is usually slower than the execution of some compiled form, so the fact that I chose to use an OCL interpreter represented a performance overhead. However, this overhead can be assumed to be more-or-less constant, in the absence of bugs in the OCL interpreter;

2. the OCL interpreter used does not implement any optimisations in the way it interprets expressions. An obvious optimisation for OCL is to cache the results of expressions that may be evaluated more than once;

3. the constraints evaluated were badly written, in the sense that no OCL interpreter could optimise their evaluation;

4. OCL may be fundamentally difficult to interpret efficiently, in that the most natural way to express a calculation for which an algorithm of manageable complexity is known to exist (having, for

example, linear complexity), may not be interpretable with the same complexity.

Of these problems, the last two are the most significant, as they affect the algorithmic complexity of evaluating a constraint. The first problem represents a constant penalty that can be improved by better engineering of the checker. An interpreter cannot be expected to perfectly optimise a constraint due to the high level of expressiveness of OCL, so although the second problem is irritating, a solution to it will not also solve problems three and four.

There is reason to believe that both problems three and four manifest themselves to some extent in this case. An example of a commonly-used, but hard to optimise constraint in OCL is the `exists` operation on collection types, which, analogously to the existential quantifier in predicate logic, searches for an element in a collection satisfying a boolean condition, and evaluates to true if such an element is found, false otherwise. The condition specified in an `exists` operation can take any form. Therefore, the design of a general algorithm by which an OCL interpreter can perfectly optimise these searches is extremely difficult, and possibly theoretically impossible. The problem is compounded if these operations are nested. Searching for a pair of elements within a collection satisfying some condition will typically have complexity $\mathcal{O}(n^2)$, regardless of what is known concerning the ordering of elements in the collection.

This particular issue can be worked around, for example by using recursive functions to implement a binary search. In my opinion, it is possible that an SLA language could be expressed in such a way that its constraints could be evaluated efficiently using my approach and the current OCL standard. However, to do so would mean avoiding a number of standard constructs provided by OCL, and would result in a language specification containing large quantities of complicated recursive functions, reducing its understandability. This assertion is as yet untested. However, if true it would imply that OCL was failing to meet an important requirement, that it should be possible to express constraints that are both easy to understand and efficient to evaluate. The capacity of OCL to meet this requirement in an SLA language, and in general to express consistency constraints that can be evaluated in a scalable manner, should clearly be the subject of future research.

The issue of maintaining large amounts of data in memory also represents a theoretical challenge. The limit on the amount of data storable could be raised by backing the repository with a database, although in cases where the amount of monitoring data was very large an even more elaborate solution might be required. However, storing monitoring data in backing storage would radically slow down its retrieval via the standard JMI interfaces. If efficient evaluation of OCL constraints were possible in theory, to remain practical a much tighter integration between the OCL evaluator and the repository would be needed, to ensure that data was accessed in a sensible order, that loading was achieved in an efficient manner and that appropriate caching of data was used to avoid delays.

4.4.3 Other runtime requirements-monitoring approaches

My approach to generating part of the implementation of a runtime requirements monitor from a language specification bears some resemblance to other efforts to embed requirements monitors in software for runtime validation of systems. Systems for this purpose typically consist of a language for expressing

the requirements, coupled with a mapping onto monitoring solutions. Representative examples are: the Java-MaC system [43] which automatically embeds monitors in Java code using a combination of byte-code rewriting and runtime libraries; and the KAOS-FLEA [25] system in which requirements specified using the KAOS methodology are monitored using the FLEA monitoring system coupled with manually implemented event detectors. These approaches are of comparable expressive power to the use of UML/OCL to describe constraints on a system. JavaMaC seems to provide extra advantages in terms of automating the instrumentation of the system, but in fact the requirements must be expressed in terms of the structure of the Java code being instrumented. The degree of abstraction at which the requirements are specified tends to determine the degree to which the placement of monitors can be automated.

A monitoring system directly related to WSML is proposed in [112]. Similar to the architecture proposed for monitoring in the WSLA specification, it consists of business management agents present at the interface to the client or the service provider or both. One of these agents is responsible for SLA monitoring and uses a specialised protocol to request monitoring data from the other agent. The agent-oriented approach taken is clearly an attempt to avoid duplication of monitoring effort, but is only appropriate under the most optimistic of assumptions regarding the trustworthiness of the parties. The SLA monitoring agent maintains a 'service model' which is a database containing SLA and scenario information as well as monitoring data. The types in this model database are described in the paper using a UML model, and the approach is therefore conceptually similar to our instantation of a JMI repository from the meta-model of SLAng. However, the schema for the model database is derived in an ad-hoc manner from the informal specification of the language semantics. It is also stated that new software components must be implemented to evaluate new types of conditions, a consequence of the reliance of WSML on semantic extensions provided by the user.

An approach similar to my own has been proposed in [53], a position paper that begins to elaborate the requirements for specifications supporting the use of contracts in an MDA process. The paper proposes that contracts can be transformed into one or more meta-models whose semantics are ultimately those of the Business Contract Language (BCL) [54], a very flexible contract definition language based on the notion of 'communities', a kind of modelling template for collaborations described in the RM-ODP. It is proposed that these models could then be processed in various ways, including implementing monitors, by tools that implement the BCL semantics. It is unclear how the transformation of contracts into these meta-models provides a benefit over simply defining a contract in BCL directly, since the expressiveness of the contract and the meta-models is likely to be equivalent. However, it is correct to identify BCL as an alternative to MOF/OCL to describe runtime requirements. In cases where requirements are primarily related to the ordering of events, BCL provides considerable semantic assistance. In more general cases, the contract-oriented nature of BCL may be hinderance to the expression of the requirements.

In response to the poor performance of the generated SLA checker, two alternative approaches to implementing monitoring solutions for the types of constraints included in various versions of the SLAng language have been tried. In [72], a monitoring solution was implemented by hand. Measurement

instruments were injected into an application server in the same manner as described in Section 4.4.1. The logic for checking conformance to throughput, reliability and latency requirements was implemented by hand, and was shown to perform adequately. In [108], a monitoring solution for timeliness and latency was implemented using timed-automata, and shown to have both good theoretical and good practical performance characteristics. Again, instruments were implemented by hand and injected into an application server. The principle objection to the approach taken in both of these cases is the need for human interpretation of the SLA specification document, which introduces an element of uncertainty as to whether the results produced by the monitoring solution are consistent with the intent of the agreement being monitored. If an automated monitoring solution proceeding from the specification of that SLA language could be perfected, this uncertainty would not be present.

In [56] the authors describe the dynamic reconfiguration of a cluster of EJB servers in response to the monitoring of performance characteristics relevant to an SLA. The monitoring system described consists of a component for determining SLA violations, another for determining whether thresholds indicative of imminent SLA violation being exceeded, and performance monitors injected into the container architecture. The SLA information used is a set of fixed parameters based on the very first version of SLAng, described in [49], and includes latency and availability parameters. The monitoring component is implemented by hand, as this version of SLAng did not benefit from a formal semantics.

4.5 Metrics for domain-specific languages

The purpose of developing a DSL is to enable certain things to be expressed. Those things might be anything at all, for example, some computer programs, some designs for bridges, accountancy data, or a catalogue of household products. The things that the DSL needs to express will have features in common, thereby delimiting the 'domain' of the language, and it is this that distinguishes DSLs from general-purpose languages, like UML class diagrams, or natural language.

Why choose to write a DSL rather than use a general-purpose language? Since UML classes can represent anything, why not just have a class diagram for everything that we want to say? The most common answer is that statements in the language are to be processed by some program. It is therefore convenient to have a language that can be easily processed, and provides some restrictions on what can be expressed to avoid authorial errors. However, from a more general utilitarian point of view three types of activities associated with the desired statements can be considered: developing the language for the statements, authoring the statements, and processing the statements automatically for some purpose. Clearly, developing a language implies a cost as opposed to choosing to use a general language, so for this to be desirable we must expect to make savings in the cost of either authoring or processing.

Much good work has been done to reduce the costs of processing DSLs, including general compiler development technologies and architectural support for processing in the form of document-models and automatically-generated meta-data repositories. This support to some extent also enables reduction in the cost of authoring statements in DSLs. Sophisticated editors require less development effort to produce and generic syntax standards such as XMI or HUTN, or languages with a common syntactic structure, such as XML, if employed may reduce the effort needed to learn a new DSL.

However, the benefits provided by these technologies can be easily eclipsed by a poorly-designed DSL. Such a language may increase costs by being hard to author, or hard to process in the sense that developing programs capable of processing the language may be costly. If the language fails to anticipate required statements then it may need extension, implying increased authoring and processing costs. The initial design of a DSL and the management of change during its lifetime is hence of commercial concern to an enterprise relying on the language, and a mature enterprise should therefore attempt to manage these processes with the help of measurements.

Starting from these assumptions, in this section I describe a novel set of metrics applicable to DSLs. I argue that these metrics provide quantitative support for qualitative judgements concerning the usefulness of a language or trade-offs in the design of a language.

4.5.1 Language specifications, extensions and statements

Any statement can be regarded as a chunk of information that must be captured in such a way that it can be recovered in the future, preferably without ambiguity. The information will be encoded using a language of some sort, resulting in an arrangement of chosen syntactic elements. To understand the syntax, and therefore recover the original information conveyed by the statement, we must refer to the definition of the language. However, the definition of the language alone does not convey the original information, because some of the information is inherent in the choice and arrangement of the lexical (or otherwise syntactic) elements constituting the statement. We can therefore observe that when expressing a statement in a language, the information burden of the statement is divided between the definition of the language and the syntax of the statement.

In the case of a DSL, it is not possible to encode all possible information into statements; information that lies outside the domain of the language cannot be encoded. However, if a DSL can express most of the information in a required statement, it might be cost-effective to reuse some of the definition of the language, by defining a language extension, as opposed to designing a new language for the desired statement, or employing a general language. In this case the information burden will be divided between the core language definition, the definition of the extension, and the syntax of the resulting statement, which uses the extended language. This division of information is depicted in the Venn diagram in Figure 4.6 in which the two-dimensional space of points represents a mapping of all concepts.

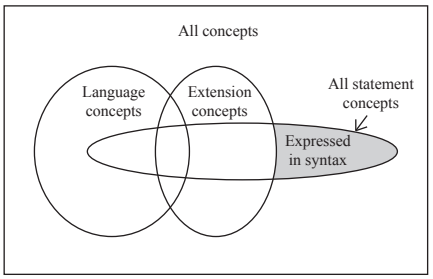


Figure 4.6: The conceptual burden of a statement is divided between the language in which it is expressed (including any extensions used) and the choice and arrangement of syntactic elements in the statement itself.

In the next section I formally define metrics that estimate how the information burden of a statement

has been distributed across the language specification, any extensions used and the syntactic structure of the statement.

4.5.2 Power, adequacy and specificity

Let us assume the following:

1. we wish to develop a language in which to make some statements;
2. we have chosen to develop a domain-specific rather than general-purpose language;
3. we have an initial set of *known statements* that we wish to make in the language;
4. statements will have a concrete representation that can be stored on a computer;
5. language specifications will have a concrete representation that can be stored on a computer;
6. we will conscience future extensions to the language;
7. language extensions will be encoded in the same manner as the specification of the language being extended;
8. we will regard languages as being freely reusable across multiple statements, whereas the syntactic part of statements will not tend to be reused. Note that this assumption may be invalidated by the common practise of providing libraries for languages. However, as I demonstrated in Section 3.2.1, for modelling languages at least, providing a library of models and providing additional language vocabulary are sometimes similar activities;
9. the cost of expressing some information in a statement, language specification or language extension is proportional to the amount of information being conveyed;
10. the measured size of a statement or language-specification is a reasonable corollary of information content, within a particular encoding scheme.

Based on these assumptions, my metrics are defined as follows:

By assumption, DSLs are reusable whereas statements are not. Therefore it is preferable to place information into a DSL whereby it may be reused, thereby reducing the cost of making statements in the future. This would tend to suggest that it is preferable to have a large DSL specification defining the syntax and semantics of relatively terse statements. I therefore define the *power* of a language with respect to a statement as being the size of the parts of the language used in the statement in proportion to the sum of the sizes of the statement and the language parts used. Given a language specification L, a statement s in the language, a function $used$ that maps a language specification and a statement to the subset of the language specification used by the statement, and a function $size$ that maps statements or subsets of language specifications to real numbers, the $power$ is hence defined as:

$$power(L, s) = \frac{size(used(L, s))}{size(used(L, s)) + size(s)} \tag{4.1}$$

Power calculations alone are not adequate to assessing the worth of a language. Consider a language required to express n different known statements, with meanings that possibly share concepts. One possible design for such a language is that each known statement is represented by a unique integer, which provides an index into the language specification in which is written separate definitions for each of the n different statements originally anticipated. Such a language is clearly powerful, but may be criticised in two ways. First, when reading any new statement, an amount of the language definition must be read in proportion to the amount of information carried by the statement, regardless of how familiar the reader is with the definition of other statements in the language, with which the new statement may share concepts. By reading, here, I also imply any effort required to develop automatic interpreters for the language. Second, any extension to the language will be unable to reuse concepts, so the extension will also be of a size in proportion to the information borne by the new statement.

The problem with this hypothetical language is that as it grows, the language definition becomes less and less specific to the individual statements that are being made. For each of the n statements, there are supporting semantic descriptions for $n-1$ other statements that are effectively irrelevant. Therefore, I define the *specificity* of a language with respect to a statement as the proportion of the size of the language elements used by the statement to the size of the language overall. Clearly a more specific language is desirable, as the cost of developing and interpreting the language definition is minimised.

$$specificity(L, s) = \frac{size(used(L, s))}{size(L)} \tag{4.2}$$

Specificity itself may come at a cost. An increase in specificity may result not only in a condensation of concepts captured by the language definition, but also in the omission of concepts. Improving specificity may mean leaving out features of the language that are anticipated as being useful but haven't found application in any currently known statement. This could be a mistake since a large class of statements required in the future may rely on these features. On encountering these statements, assuming that it is cost-effective to do so, it will be necessary to rely on extensions to implement the missing concepts. It could be argued that defining these extensions is no more expensive than simply implementing support for the concepts in the first place, so this is not an additional cost. However, if the extensions are not then incorporated into the original language, perhaps because the original language has become standardised, or because it is deemed that the extension would unduly harm the specificity of the original language, there is the danger that a similar extension may need to be defined again in the future, resulting in a repeated cost.

It is therefore relevant to ask: to what extent is a language adequate to a new statement? If the new statement will require an extension to the language to enable its expression, then the language is clearly not completely adequate, and the relative size of the elements used from the language, and the size of the elements used from the extension indicate the magnitude of the contribution of the language to the statement. Given, in addition to the language, a language extension E, I state that the `adequacy` of a language with respect to a statement is given by:

$$adequacy(L, E, s) = \frac{size(used(L, s))}{size(used(E, s)) + size(used(L, s))} \tag{4.3}$$

Note that in each case I have defined power, specificity and adequacy in terms of a single statement. Clearly, measures for average power, specificity and adequacy over sets of statements can also be obtained and used to assess the quality tendencies of a language over several statements.

Measurements are principally useful for comparisons between various subjects, so the metrics as proposed may be helpful when evaluating candidate languages or language-design choices. What constitutes a good design for a DSL will remain a largely subjective matter and present a considerable challenge to empirical investigation in the future. However, valuations of my metrics could reinforce the following subjective judgements:

- a language with a low average power is likely to be more expensive to use than a language with a higher power for the same statements. The extra cost is in preparing larger statements;
- a language with a low average specificity is likely to be more expensive to use than a language with a higher average specificity for the same statements. The extra cost will be in interpreting the more redundant language specification;
- a language with a low average adequacy for a set of statements is likely to be more expensive to use than a language with a higher average adequacy for the same statements. The extra cost will be in defining extensions to the core language;

From the point of view of evolving DSLs the objective of improving measurements based on the metrics may also be used to suggest candidate changes to a particular language. Clearly a language designer should seek to improve values for each of my metrics, therefore hoping to reduce costs, but must do so without invalidating the assumptions on which the metrics are based (e.g. that the cost of preparing a statement is proportional to its size – it might be that a language change that improves the power of the language makes statements unfeasibly difficult to formulate), and without compromising too drastically on one measure in favour of another. Possible courses of action are:

- to improve the power of a language by introducing more specialised constructs, with greater semantic refinement. This may have the effect of reducing specificity since the constructs will only be usable in certain statements;
- to improve the specificity of a language by combining constructs which share a conceptual basis. This has the potential to reduce the power of the language by requiring subtle distinctions between concepts to be discriminated in statements;
- to improve the specificity of a language by removing infrequently used constructs. This may reduce the adequacy of the language by obliging the use of extensions for certain statements;
- to improve the adequacy of the language by introducing new constructs for concepts commonly encountered in extensions to the language. This may reduce specificity by introducing constructs that will only be reusable in certain statements.

All of the metrics rely on the definition of two functions $size$ and $used$: $size$ is a mapping from the concrete representation of a language specification, extension or statement to the range of real numbers; $used$ is a mapping from the concrete representation of a language specification or extension to the concrete representations of those parts of the specification or the extension upon which a statement depends.

The definition of both mappings are dependent on how language specifications and statements are encoded. According to the assumptions upon which the metrics are based, the cost of specifying some information is proportional to the quantity of the information being specified, and the size of a statement or language specification is proportional to the amount of information they encode. The $size$ metric, upon which the definition of my metrics depend, takes a statement, or all or part of a language specification or language extension, and maps it to a real number. For the economic interpretation of the metrics to remain valid, it is therefore important that for a particular set of statements, language definition and extension, the $size$ function is defined in such a way that the values that it returns are proportional to the information encoded in the artifact that it is measuring, despite the fact that language specifications and statements may be encoded differently. Clearly the $size$ metric should increase monotonically with the amount of information represented by an artifact.

If the size function tends to overestimate the effort required to define the language, in comparison to a statement, or vice versa, then it may be biased, and measurements for power will therefore be misleading. Bias of this kind is less problematic when considering specificity, as only the language is being measured. This is also the case for adequacy, as I have assumed that extensions are encoded according to the same scheme used to encode to core language. The design of the $size$ metric becomes more difficult when attempting to compare languages encoded according to different schemes, as then bias between the encoding schemes must be considered.

In the next section I discuss how the $size$ and $used$ functions can be defined for DSLs with meta-models defined using the EMOF model.

4.5.3 Defining $size$ and $used$ functions for EMOF and OCL-based languages

Previous work by other authors (discussed below) has highlighted the possibility of defining metrics for modelling languages using extensions to the meta-models of those languages. In particular, additional measurement classes are defined with properties typed to refer to the model-elements being measured. Side-effect-free operations are then defined on these types using the Object-Constraint Language (OCL) to calculate values for the metrics. According to the four-layer meta-modelling architecture, discussed in Section 3.1.3, the procedure for calculating measurements of M1 elements is described in terms of their M2 types.

This approach is ideal for providing a formal definition for language-specific metrics, assuming a meta-modelling language incorporating OCL. Coupled with a standards-compliant JMI generator and OCL interpreter, such formal specifications are adequate input for the automatic generation of a tool capable of calculating metric values from models.

However, without modification the approach is not ideally suited for metrics that are both language-

independent, and involve the comparison of statements and language specifications, or in the terminology of the MDA, models and meta-models. Power, adequacy and specificity are of this type.

However, providing a common language for encoding language specifications can be assumed, language-independent metrics may be defined using the same approach, but by embedding the definition of metric types in the meta-meta-model. If EMOF and OCL are used, the metric types can be defined in extensions to the EMOF meta-model using OCL. This provides the opportunity to reason about the types of objects present in meta-models, and hence compare languages. This is placing metric definitions at the M3 layer to reason about the M2 layer.

The residual problem is defining metrics that describe the comparison of languages (at the M2 layer) with the models (or statements, at the M1 layer) that are expressed. To achieve this using OCL requires an apparent violation of the four-layer meta-modelling architecture, since OCL evaluation (according to the OCL 2 standard) only acts on a model from a single layer. However, by observing that meta-model syntax and model syntax may both be modelled in an abstract, object-oriented fashion, this distinction can be avoided.

Stated otherwise, consider that models conform to meta-models according to the semantics of the meta-modelling language, which may be defined at the M3 layer using the model-denotational approach of associating the model of the syntax of the meta-modelling language with a model of its semantic domain. The MOF 2.0 standard takes this approach and defines an 'abstract semantics' for the CMOF model. The types used in these semantics are apparently defined at the M3 layer, but instances of these types must correspond to model elements at the M1 layer. This apparent contradiction highlights the artificiality of the four-layer meta-modelling paradigm.

In my approach to defining a $size$ metric for models and meta-models, I make use of the model-denotational approach to define a similar semantics for the EMOF model (regrettably no standard semantics are available) along the same lines as that provided for CMOF. This semantic model is used to extend the syntactic model of EMOF, with associations between meta-model and model types representing type-instance relationships. The semantic model for EMOF is shown in Figure 4.7.

Note that both models and meta-models can be regarded as instances of the types in the semantic model, because both models and meta-models are instances of types defined according to the EMOF model. In the case of meta-models, these instances represent the types in the EMOF model itself, because EMOF is defined recursively.

Because models and meta-models can be represented using the same types, it is possible to define a $size$ metric precisely, which can be applied equally to both models and meta-models and therefore plausibly reduces bias in measuring the size of each. Moreover, the joint model of EMOF syntax and semantics can be compiled into a JMI repository capable of containing any combination of model and meta-model, without recompilation. Models and meta-models can be loaded into such a repository and used to calculate metrics. Some additional coding is required to import a model as instances of the semantic types, since a standard XMI reader will expect to be instantiating classes generated from the model's meta-model, rather than the semantic types.

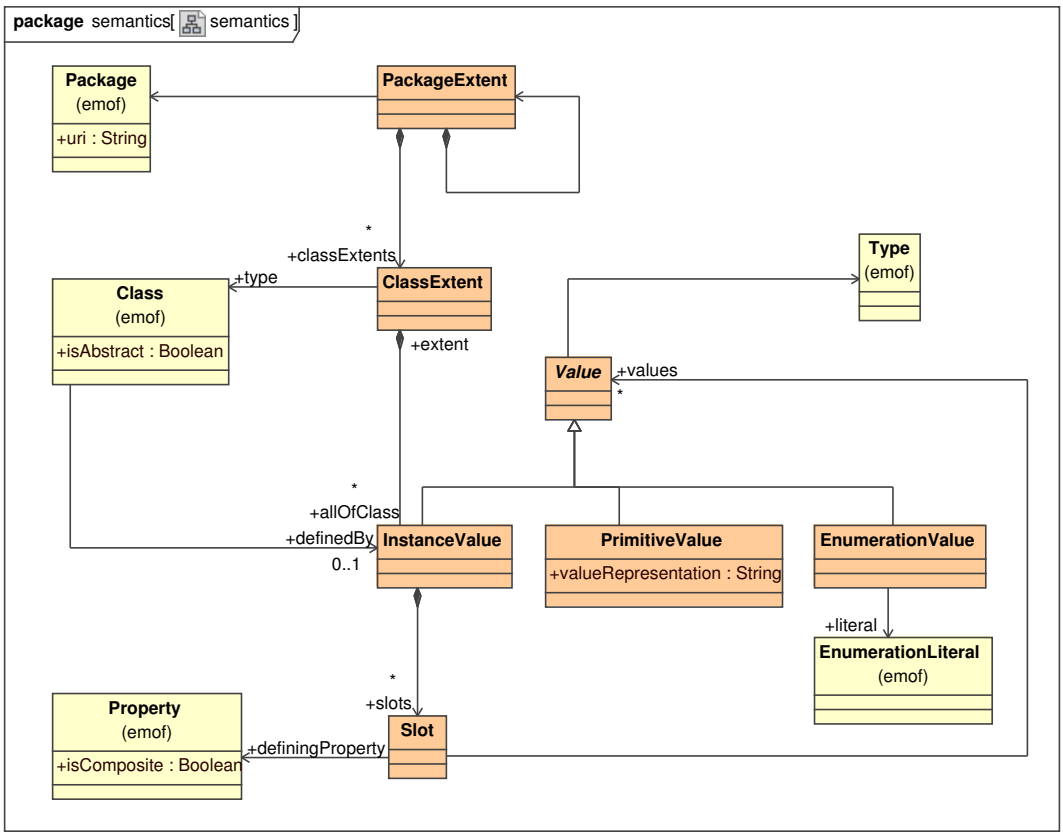


Figure 4.7: A semantic model for the EMOF language

Model instances, according to my semantics for EMOF, consist of a number of different types of elements. These are: package extents; class extents; instance values, consisting of a number of slots, each containing one or more values; primitive values, which may be strings, integers, reals or booleans; or enumeration values, which identify a literal of an enumerated type.

Package and class extents exist for each of the packages and class-types defined in the meta-model for a model. They are automatically generated in a repository for the purpose of navigating the values in the model, and do not express any meaning in a model. I therefore argue that they do not contribute to the size of the model.

The values of different types are clearly the result of effort spent specifying the model. The question is, what effort? If we assume that the effort specifying the existence of any two instances is the same, which I implicitly have by assuming that effort is proportional to model size, then we still need to determine the relative effort involved with specifying primitive values, including strings, whose length may vary, the relative sizes of instance values for which property values have been specified, and the contribution to model size of nulls, which may be interpreted as either not specifying a property (leaving it with an unknown value) or asserting that it has no value.

With these considerations in mind, I propose that any size metric for models should be parametric, providing the opportunity for the user of the metric to specify weights for each of the elements corresponding to their assessment of the distribution of effort.

My $size$ metric is therefore defined as the sum of all weighted elements in a model, possibly taking into account the length of strings. Elements that the user chooses to omit may be weighted 0. The $size$ metric is therefore defined as an additional class in the EMOF syntactic and semantic meta-model as follows (this definition relies on a non-standard extension to EMOF that disambiguates the identification of the type of a primitive):

```
class ModelSizeMetric {

    instanceWeight : ::emof::Real
    referenceWeight : ::emof::Real
    enumerationWeight : ::emof::Real
    nullWeight : ::emof::Real
    integerWeight : ::emof::Real
    booleanWeight : ::emof::Real
    realWeight : ::emof::Real
    stringWeight : ::emof::Real
    stringElementWeight : ::emof::Real

    instanceSize(instanceValue : ::semantics::InstanceValue) :
        ::emof::Real = {

        instanceWeight +
        instanceValue.slots->collect(s : ::semantics::Slot |

            if s.values->size() = 0
            then nullWeight
            else s.values->collect(v : ::semantics::Value |

                if s.definingProperty.type.oclIsKindOf(
```

```
                    ::extensions::OCLEquivalentPrimitiveType)
                then
                    let primitiveTypeKind =
                        s.definingProperty.type.oclAsType(
                            ::extensions::OCLEquivalentPrimitiveType
                                ).kind
                    in
                    if primitiveTypeKind =
                        ::extensions::OCLEquivalentKind.STRING
                    then
                    stringWeight +
                    v.oclAsType(::semantics::PrimitiveValue
                        ).valueRepresentation.size() *
                        stringElementWeight
                    else
                    if primitiveTypeKind =
                        ::extensions::OCLEquivalentKind.INTEGER
                    then
                    integerWeight
                    else
                    if primitiveTypeKind =
                        ::extensions::OCLEquivalentKind.REAL
                    then
                    realWeight
                    else
                    booleanWeight
                    endif endif endif
                else if s.definingProperty.type.oclIsKindOf(
                    ::emof::Enumeration)
                then enumerationWeight
                else
                    referenceWeight +
                    (if s.definingProperty.isComposite
                    then instanceSize(
                        v.oclAsType(::semantics::InstanceValue))
                    else 0.0
                    endif)
                endif endif
            )->sum()
            endif
        )->sum()
    }
}
```

The joint meta-model also provides the opportunity to define a useful $used$ mapping. Since instances are associated with the types to which they conform, it is straightforward to determine the set of meta-model types used directly by any statement. Unfortunately, it is not totally straightforward to then measure the size of these types, since the $size$ metric defined above relies on having instance objects, not type objects to measure. However, a minor additional extension to the meta-model, also shown in Figure 4.7 allows types in meta-models to preserve a link to instance-objects representing them. A small amount of additional programming adds functionality to the repository to convert any meta-model types into instance objects referring to an instance of the EMOF model, and sets the `definedBy` properties for the meta-model types with references to these instances. A given instance object explicitly uses one

type to define its own structure, and one type for each of its attributes. It is debatable whether the types for null attributes should be considered to be used, so this can be a parameter of the definition of $used$ chosen. EMOF types may inherit properties from other types, so super-classes of explicitly used types should also be considered to be used. For a given instance object, $used$ may therefore be defined by the following side-effect-free operations, declared on the type `InstanceValue`:

```
typesUsedExplicitly(countNullAttributeTypes :
    ::emof::Boolean) :
    ::emof::Type[*] unique = {

    Set(::emof::Type) { type }->union(

        slots->collect(s : ::semantics::Slot |

            if s.values->notEmpty()
            then
                if s.definingProperty.isComposite and
                    s.definingProperty.type.
                        oclIsTypeOf(::emof::Class)
                then s.values->collect(v : Value |
                    v.oclAsType(InstanceValue).
                    typesUsedExplicitly(
                        countNullAttributeTypes))->asSet()
                else Set(::emof::Type) { s.definingProperty.type }
                endif
            else
                if countNullAttributeTypes
                then Set(::emof::Type) { s.definingProperty.type }
                else Set(::emof::Type) { type }
                endif
            endif
        )
    )->asSet()
}

inheritanceClosure(explicitTypes : ::emof::Type[*] unique) :
    ::emof::Type[*] unique = {

    let moreExplicit = explicitTypes->collect(
        t : ::emof::Type |

        let setOfT = Set(::emof::Type) { t }
        in
        if t.oclIsKindOf(::emof::"Class")
        then
        setOfT->union(
            t.oclAsType(::emof::"Class").superClass->asSet())
        else
        setOfT
        endif
    )->asSet()
    in
    if moreExplicit = explicitTypes
    then explicitTypes
    else inheritanceClosure(moreExplicit)
    endif
```

```
}

used(countNullAttributeTypes : ::emof::Boolean) :
    InstanceValue[*] unique = {

    inheritanceClosure(typesUsedExplicitly(
        countNullAttributeTypes)
        )->collect(definedBy)->asSet()
}
```

Unfortunately, this definition of $used$, although helpful, does not represent a definitive solution for defining this function for languages specified according to my approach. Languages that benefit from a model-denotational semantics will contain semantic meta-model classes that are never directly used by a statement, but nevertheless provide an important contribution to defining the meaning of the language. The structure of the meta-model may be used to some extent to determine which of these semantic classes are relevant to a particular statement, but this is not completely satisfactory as it fails to discount types that are only relevant depending on the value of the statement. Perfect reasoning for the usage relationship would require the hypothesis of semantic structures consistent with the constraints on the syntactic types used directly by a statement. The success or failure of such reasoning with OCL is almost certainly undecidable in general due to the high expressive power of OCL. Therefore, evaluating the $used$ relationship for statements in EMOF-defined languages remains a matter of human judgement.

An implementation of this meta-model and all supporting code is available as part of the UCL MDA tools [135].

4.5.4 Related work in metrics

The study of metrics is a large field. Prior work particularly related to my own falls into several categories: the specification of metrics for modelling languages; the validation of metrics; metrics for object-oriented systems in general; metrics for modelling languages; and metrics for software reuse.

In the first category, several approaches have been specified for defining metrics for modelling languages. The formalisation of object-oriented metrics over UML models using side-effect-free operations defined on classes in the UML meta-model was first proposed by [8]. The minor additional generalisation of introducing metrics classes (as I use for the size metric in the preceding section) was first proposed by [62]. The authors of this second work also describe in [61] the definition of metrics over the Dagstuhl Middle Metamodel [52], a model representing the union of features from a number of object-oriented languages and intended to support refactoring activities that involve translation between languages. [61] establishes the feasibility of defining metrics over a meta-model other than the UML meta-model, and also describes the automated generation of a repository incorporating Java code compiled from the OCL expressions embedded in the meta-model to calculate the metrics.

Other approaches to defining metrics for modelling languages have also been proposed. [64] proposes a set of object-oriented metrics that are defined as specialisations of operations on graphs. These metrics have the advantage of being formally defined. Also the underlying graph formalism is compatible with several object-oriented formalisms, so the authors argue that the metrics are to some extent language independent. However, to fully understand how the metrics apply to any given language, it

is necessary to first defining a mapping from the structure of the language to the graph structure. This is inconvenient, and there is no way of determining whether any two mappings from two different languages produce results that are in any way equivalent, so it is not clear that calculations over graphs for statements in different languages can be compared in this case. However, since the metrics defined are primarily counts of elements of various types, this may not be problematic.

[136] proposed the use of a model transformations language, specifically ATL [42] to define metrics, effectively transforming a model into another model representing the results of metric calculation. Although ATL has elements with both an imperative and declarative character, this alternative proposal can be seen as representative of a division within modelling research between those that prefer to describe actions performed on models, and those that prefer to describe relationships between models.

[31] describes a meta-model for a metrics-definition language. A graphical syntax allows the definition of metrics and taxonomies of metrics. The semantics of the metrics may either be defined using expressions based on a fundamental set of metrics that assume that a language has an object-oriented abstract syntax, or may be coded using Python [107], assuming access to a model repository generated by the tool AToM [67]. The approach is essentially tool specific, since AToM uses a non-standard meta-modelling language.

Validation of metrics is the process of determining whether the tendencies of a metric correspond to the tendencies of the real world quantity that they are presumed to measure. A useful survey is [39]. What would be required for power would be a demonstration that statements in a more powerful language tend to be cheaper to author than statements in a less powerful language for the same purpose, since the cost-effectiveness of authoring a statement in a given language is the real-world property that power is attempting to measure. Similar properties would need to be proven for specificity and adequacy. Naturally, such validation would require empirical studies outside the scope of this work, particularly given that producing languages and models are time-consuming and expensive activities. I have therefore relied on an informal justification of my metrics. However, if the metrics become widely used to aid decision of real commercial value, such validation may become cost-effective.

Since meta-models are typically object-oriented, it is reasonable to consider what benefit may be derived from applying traditional object-oriented metrics to their measurement. The classic suite of object-oriented metrics were proposed by Chidamber and Kemerer, and are well-known as the CK metrics [13]. [60] offers OCL definitions of these metrics. The problem with the use of these metrics is that it is highly debatable whether knowing such values for such properties as weighted-methods per class, or depth of inheritance tree allows any useful conclusions as to the quality of a meta-model to be derived. However, in [58], the authors showed the use of simple object-oriented metrics to assess two quite useful properties of the evolving UML meta-model: first, by measuring the change in quantity of various types of meta-model element, the authors were able to produce a composite metric that reasonably corresponds to a subjective judgement of the relative dissimilarity between consecutive versions of the language; second, by calculating standard object-oriented metrics and producing weighted composite metrics for reusability, flexibility, functionality, extendibility and effectiveness, they argued that the

design of the UML meta-model is tending to improve as the version number increases. These composite metrics have a similar character to my metrics, but attempt to measure different qualities of a language. However, their subjective value is encoded into the weighting of the more primitive object-oriented metrics from which they are calculated, and does not have the justification of a straightforward economic argument, so they may be harder to validate.

Finally, since my metrics may be interpreted as measuring the reusability of a language-specification, a comparison with metrics for software-reuse is possible. A good survey of this field is [27]. Here some direct analogies are possible. What I have defined as power is closely equivalent to a product-reuse percentage, which is the ratio of reused code in a product to the combined size of reused code and new code. Specificity is equivalent to a quantity of reuse metric for a library or program, namely the ratio between reused code and total code. There seems to be no direct analogy for adequacy, since reuse metrics only deal with two categories of artifact, the new and the reused, where as adequacy relies on three things, a language, an extension and a statement. Moreover, all of my metrics seem to differ from the reuse metrics by being specific a particular usage of a language, rather than to the static properties of the language. In this sense they are more similar to measuring executed code using runtime monitoring. Nevertheless, the similarity between reuse metrics and my metrics suggests that other reuse metrics may have useful analogies in modelling languages.

4.6 Summary

In this chapter I have presented a number of research contributions related to language specifications, which I have defined as being a single definitive point of reference concerning the syntax and semantics of a language. In particular I have focussed on the type of language specification that will naturally arise from following the recommendations developed in the preceding chapter, with respect to the definition of a domain-specific language for ASP SLAs. Such a language, and languages defined along the same lines, will have an abstract syntax, one or more concrete syntaxes, and a description of its semantics. The abstract syntax will be defined using an object-oriented model. The semantics will be defined using a combination of the model-denotational style, which relies on an object-oriented domain model, and natural language statements.

I described two pervasive problems with the approach used by the OMG to document languages of this kind: first, definitive language specifications tend to be defined in documents that are predominantly human-readable, and not amenable to automatic processing. Although formal models of some languages are available, or in some cases can be inferred, these models are not regarded as definitive, and nor are they adequate to define a language, as they typically do not contain sufficient semantic documentation; second, although concrete statements in these languages may in some cases reference a formal description of the syntax of the language in which they are defined, the link between the statement and the definitive language specification is usually non-existent. This is almost certainly due to confusion concerning whether the human-usable documentation of the language or the formal description of the syntax should be regarded as being definitive.

I also observed that these problem compounded the difficulties faced when dealing with exten-

sible languages, in which extensions must regularly be defined, documented and combined with core languages.

To address these issues I proposed a raft of relatively minor changes to core OMG standards, in particular the MOF, XMI and HUTN standards. The MOF should be changed so that definitive human-readable semantic documentation can be included in meta-models, and these meta-models can then fulfil the role of language specifications. They will be amenable to automated processing, can be used to provide context-sensitive assistance, and can easily be combined with the specifications of extensions to produce new combined language specifications. In addition, the relevant concrete-syntax standards should mandate the inclusion of links not only to themselves, but to the language specification being used in any concrete statement, thereby ensuring that a statement can be interpreted according to the definitive documentation of the language in which it is written. I described the UCL MDA tools, which implement these recommendations. I compared the UCL MDA tools to available alternative MDA tools.

I next described the potential for tools such as the UCL MDA tools to be used to test language specifications by evaluating OCL constraints. In the case of ASP SLA languages, this type of evaluation can also be used to check conformance of a model of service provision to the conditions included in ASP SLAs, as part of a runtime monitoring system. I described an initial attempt to construct such a system using an early version of the SLA language SLAng, the latest version of which is discussed in more detail in Chapter 6. Although easy to implement, this system was impractical due to the computational complexity associated with evaluating OCL constraints. This problem may be the fault of the OCL interpreter or the formulation of the constraints, and addressing it remains the topic of future work. I discussed this work in the context of other work to provide runtime monitoring solutions.

Finally, I described novel set of metrics, intended to be helpful in the evaluation and evolution of domain-specific languages. These metrics, called *power*, *adequacy* and *specificity*, rely on the observation that the information conveyed by a statement in a DSL, and therefore, by my assumption, the authorial effort required to convey that information, is divided between the choice and arrangement of lexical elements constituting the statement, and the description of the language in which the statement is written. I discussed how these metrics might indicate problems in a DSL, and how their values may be improved by redesigning the language. I described how the calculation of these metrics could be implemented in an unbiased way for languages defined according to my language-specification approach. I discussed the metrics in the context of previous work on the measurement of meta-models, and more generally of object-oriented systems. The metrics are used to assist in the evaluation of SLAng, in Section 8.3, and to demonstrate its evolution, in Section 8.4.

Chapter 5

The Monitorability of ASP SLAs

In Chapter 3 I have described an approach to defining domain-specific languages for SLAs that has, amongst other advantages, the potential to specify SLA languages that are precise, and hence express SLAs with a precise meaning. Precision is one of two fundamental requirements for SLAs that stem from the requirement that an SLA be protectable, in the sense that any disagreement concerning the SLA should be resolved according to the original intent of the agreement. Precision is necessary so that an unambiguous intent can be retrieved from the agreement in the event of a dispute. The other necessary condition for protecting the agreement is that reliable evidence can be collected and presented that is relevant for determining how the intent should be applied, and which is convincing to the parties involved in the dispute. Depending on how an SLA is written, such evidence may be more or less easy to obtain. I refer to the property of an SLA that determines how easily relevant and trustworthy evidence may be obtained as its *monitorability*.

In Section 2.6 I have discussed the fact that to mitigate the risks of all of the parties in a service-provisioning scenario, a system of several SLAs may be required. In this chapter I introduce a technique for analysing systems of SLAs to determine the degree of monitorability possible, according to a classification of monitorability that I describe.

I apply this technique to identify the most monitorable system of SLAs capable of insuring timeliness constraints (a common requirement in the ASP scenario) for the three-role Application-Service Provision (ASP) scenario, described in Chapter 2.

The system contains SLAs that are at best mutually monitorable, and of all the possible systems of SLAs that might be established in the scenario, only a single system of SLAs achieves this level of monitorability. I also show that this level of monitorability is possible for electronic-service provision scenarios involving networks administered by multiple ISPs, although I do not prove that no higher level is possible.

This result has several significant implications:

First, the system of SLAs that achieves mutual-monitorability in the three-role scenario is not one that is in common usage. I discuss the practical implications of this further below.

Second, all SLAs in an identified system of SLAs require only guarantees related to the behaviour of the service as experienced at a certain point in the network, not end-to-end guarantees. This implies that a language supporting only mutually-monitorable SLAs in the ASP scenario will not need to address

the provision of network-services, only the provision of electronic-services and the real-world behaviour of an application service.

Finally, SLAs that are at best mutually-monitorable imply the potential requirement for reconciliation of monitoring data between the parties, and hence the need to constrain the parties to report honestly. Honest reporting is made more difficult both to achieve and to assess by the inevitable presence of measurement error in any measurement of a physical system. Since true values of physical quantities can never be known for certain, this implies the need for a constraint that bounds error in reported values, and which can be approximately monitored, in the sense that confidence in the belief that the constraint has been either met or violated can be obtained, even if this can never be known for certain. Therefore, I also describe a constraint on the precision and accuracy of reported measurements, and its approximate monitorability using a statistical hypothesis test.

The material in this chapter is largely reproduced from [120]. Jason Crampton assisted in the formalisation of the monitorability model. Allan Skene assisted in demonstrating the approximate monitorability of the accuracy constraint.

5.1 Monitorability

Three parties participate in the basic ASP scenario introduced in Chapter 2: the client, the service-provider, and the network service provider. Discounting for the present the real-world behaviour of the service, and assuming that the interface to the service is a simple, synchronous electronic-service, let us consider what could go wrong for the client in this situation.

One possibility is that having submitted a request, no response is received by the client within some reasonable interval of time. The client complains to the service provider that a timely response was not received. The provider claims that no request was received, produces a log of requests as evidence supporting this claim, and directs the client to complain to the ISP who was responsible for conveying the request to the service. The ISP insists that the request reached the service provider and produces a log supporting this claim. Who can the client trust? Both the ISP and electronic-service provider have delivered easily fabricated evidence concerning an event, the delivery of the request at the service-provider's interface, that the client was incapable of independently monitoring.

Let us assume that for their own reasons, the client chooses to mistrust the service provider, and requests that they enter into a service-level agreement. In this agreement the client seeks to reduce the costs that they expect to incur when the service fails to perform as expected, by receiving a penalty from the provider, also giving the provider a disincentive to poor performance. The client perceives that the problem with the service is a lack of availability due to an erratic maintenance regime on the part of the provider. The provider duly commits to provide 95% availability over the lifetime of the contract of which the SLA forms a part.

Over the period of the contract, the client uses the service frequently and frequently responses are not generated following requests. At the termination of the agreement the client seeks compensation from the provider, who refuses to pay. The provider argues that although the service was unavailable when requests were made for which responses were not received, at all other times the service was

available. Accumulating the microscopic intervals during which the (numerous) failed requests were being delivered and the service was admittedly unavailable still does not amount to 5% of the lifetime of the contract, and hence the provider need not pay. In this second example, the client has entered into a agreement which is, possibly implicitly, defined in terms of events that the client cannot directly observe, namely the passing of the service from an available to unavailable state, and vice versa. Since the client cannot observe these events, it must take the word of the service provider with respect to the availability of the service, or else pursue compensation with very little support from the original SLA.

These examples highlight monitorability as an important requirement for SLAs. In both cases, the client became concerned with an event that they fundamentally could not observe: in the first example, the delivery of the request to the service; in the second, the transition of the service between availability states. In both cases, the concern arose because another event that they could observe, the delivery of the response to the client, failed to occur when expected. Had the client complained about this latter event, they would have had a stronger argument, because no party could convince them of a falsity concerning the event in question.

If the client complained about the quality of the service in relation to the delivery of responses, to whom should they complain? Without being able to monitor events within the network and service, it will not be apparent to the client which party, the ISP or service provider, is responsible for poor performance. Neither ISP nor service provider may wish to take responsibility for the overall QoS delivered to the client when the actions of the other could cause any constraints on the QoS to be violated. On the other hand, it may be possible for one of the service providers to mitigate this risk by obtaining an SLA from the other. In this case the monitorability of the second SLA must also be considered.

Considerations of this kind for a particular scenario beg the question: is any system of SLAs possible to guarantee a particular requirement in which all SLAs are monitorable, and represent acceptable risks to the providers? In the next sections I present an abstract mathematical model of such scenarios and describe how it may be refined and permuted to answer this question, and provide other insights, for a particular scenario. An analysis of the monitorability of systems of SLAs containing timeliness constraints over electronic services is used both to gain insights into the requirements for languages expressing such SLAs, and demonstrate the use of the monitorability model.

5.1.1 Modelling systems of SLAs

I assume that two or more parties participate in some form of interaction comprising a sequence of actions each performed by one of the participants. Examples include electronic service provision, resource provisioning in virtual organisations, etc. Particular parties may have certain expectations about the execution of particular actions. For example, a party may specify a requirement that these actions are executed within a certain amount of time, as in the case of service provisioning. Another participant may agree to pay penalties to this first party if these requirements are not met.

Definition 1 *An* interaction *between two or more entities belonging to a set of* participants P *is modelled as a sequence of* actions*,* A*. Each action* a *is associated with an* actor $\alpha(a) \in P$*, the party that may perform the action.*

Figure 5.1 depicts the interaction model for the electronic service scenario. Interactions with the real world and any database have been elided, and a simple synchronous model of communications has been assumed. Formally, the model is expressed as follows:

- in the three party scenario: C is the client, I the ISP and S the service provider; $P = \{C, I, S\}$;
- the client occasionally takes action to invoke the service by dispatching requests into the network;
- the ISP conveys requests through the network, eventually delivering them to the server (operated by the service provider);
- having received a request, the server performs some service and injects the result back into the network;
- the ISP is then responsible for the delivery of the result to the client.
- the actions are $A = \{dispatch, send, process, respond\}$, and responsibility is allocated as follows: $\alpha(dispatch) = C$, $\alpha(send) = I$, $\alpha(process) = S$, and $\alpha(respond) = I$.

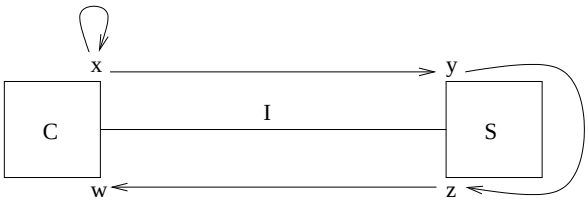


Figure 5.1: An interaction model for application service provision showing actions and their associated events

Definition 2 *An action a may give rise to any number of* events $\epsilon(a) \subseteq E$*, where for all pairs of actions a and a', $\epsilon(a) \cap \epsilon(a') = \emptyset$ if $a \neq a'$.*

The events modelled for a given scenario depend on the requirements of interest to the parties. I now restrict my analysis of application service provision to systems of SLAs governing the timeliness of service provision. I therefore define events corresponding to the completion of each of the events in the scenario $E = \{x, y, z, w\}$ in relation to our actions such that $\epsilon(dispatch) = \{x\}$, $\epsilon(send) = \{y\}$, $\epsilon(process) = \{z\}$, and $\epsilon(respond) = \{w\}$. These events are chosen because service provision should be deemed to begin when a request has been fully submitted, and end when a response has been fully received. It is easy to see that this simple model meets the requirement that no event is the result of more than one action.

Monitorability analysis using the model will be supported by the notion that particular events are only visible to a subset of the parties in the interaction:

Definition 3 *Each event $e \in E$ has a set of* observers $\rho(e) \subseteq P$*, the parties that may observe the event. It is assumed that for all $e \in \epsilon(a)$, $\alpha(a) \in \rho(e)$.*

The events in our model occur at network interfaces, so I define the observers for each event as follows: $\rho(x) = \rho(w) = \{C, I\}$, and $\rho(y) = \rho(z) = \{I, S\}$. Naturally, each event is visible to the actor responsible for that action that causes it.

I now describe a model for requirements over events:

Definition 4 *Events may have attributes. A set of* observations*, O, may be defined over these attributes, with each proposition $o \in O$ pertaining to a subset of events $\pi(o) \subseteq E$. These observations can be considered to be logical predicates concerning the values of attributes of observed events.*

Latency constraints in SLAs will potentially place restrictions over the timing of the events x, y, z and w. I now state that occurrence time is an attribute of the events, and define observations that capture the requirements of the parties and the constraints that we may wish to include in SLAs.

Specifically, I wish to express the client's requirement that w occurs within some interval following x. I denote the observation that this occurs in abstract as $w - x < t$, where t may take any positive, non-zero value.

This constraint can potentially be achieved by constraining the relative times of events occurring between w and x, the intuition being that the overall time taken to complete a request is acceptable if the times taken to complete each action required to service the request are also acceptable. For example, $w - x < t$ will hold if $y - x < t_1$, $z - y < t_2$ and $w - z < t_3$, in all cases where $t_1 + t_2 + t_3 < t$.

Even supposing that the delay between events x and y exceeds the arbitrary bound t_1, this does not imply that the client's overall requirement will be violated. $w - x < t$ will be met if $z - x < t_1 + t_2$ and $w - z < t_3$, and other combinations of constraints are also possible. The total set of observations with which we will be concerned is hence $O = \{y - x < t_1, z - y < t_2, w - z < t_3, z - x < t_1 + t_2, w - y < t_2 + t_3, w - x < t\}$, where $t_1 + t_2 + t_3 < t$ and t, t_1, t_2, and t_3 are all positive and non-zero.

The mapping π from observations to the events over the attributes of which the observations are defined is obvious in this case from the naming of the observations, e.g. $\pi(y - x < t_1) = \{y, x\}$.

Clearly, if observations are predicates over a scenario, then the truth values of observations are potentially related. In the ASP case, we can state that $w - x < t$ will always hold if $y - x < t_1$, $z - y < t_2$ and $w - z < t_3$ hold, amongst other relationships.

Definition 5 *Observations are related by an* entailment relation*, $\models \subseteq 2^O \times O$, where $U \models o$ is interpreted as stating that o is always true when all observations in U are true.*

To more conveniently describe relationships between observations, a minimal dependency mapping D may be defined based on the entailment relationship. I say $U \subseteq O$ entails observation o if $U \models o$ and I say U is *minimal with respect to o* if for all $U' \subset U$, $U' \not\models o$. Hence $D(o) = \{U \subseteq O - \{o\} : U \models o$, and U is minimal with respect to $o\}$.

In our scenario, the dependency mapping may be enumerated as follows:

$$
\begin{aligned}
D(w - x < t) &= \{\{y - x < t_1, z - y < t_2, w - z < t_3\}, \\
&\quad \{z - x < t_1 + t_2, w - z < t_3\}, \\
&\quad \{y - x < t_1, w - y < t_2 + t_3\}\} \\
D(z - x < t_1 + t_2) &= \{\{y - x < t_1, z - y < t_2\}\} \\
D(w - y < t_2 + t_3) &= \{\{z - y < t_2, w - z < t_3\}\} \\
D(y - x < t_1) &= \emptyset \\
D(z - y < t_2) &= \emptyset \\
D(w - z < t_3) &= \emptyset
\end{aligned}
$$

Definition 6 *A party $c \in P$ may impose certain* requirements *on the execution of a sequence of actions. The pair (c, o) denotes a requirement by c that o should hold. I denote the set of requirements by $r \subseteq P \times O$.*

The client's latency requirement is the fundamental requirement addressed in this analysis. Hence $(C, w - x < t) \in R$.

Definition 7 *A party p can provide an* SLA *to insure any requirement. The SLA states that the party with the requirement will receive compensation from p if the requirement is not met. SLAs are modelled as a pair $(p, (c, o)) \in L$ where L is the set of SLAs in a particular scenario and (c, o) represents a requirement.*

For example, I could offer C an SLA matching C's requirement: $(I, (C, w - z < t_3))$. Any combination of parties with requirements is possible, so for example, C might also offer S an SLA of the form $(C, (S, w - y < t_2 + t_3))$.

Having now established definitions for SLAs and the observability of events in our model, I now define the levels of monitorability possible in a given model.

Although parties may not be able to observe an event themselves, they may have the event reported to them by a party that they trust. However, I suggest the conservative restriction that a party p should not be trusted to report on an event if p has a financial interest in that event. A party has a financial interest in an event if they provide or receive an SLA that insures an observation to which the event is pertinent. The interest arises from the desire of the client to receive penalty payments, and the desire of the provider to avoid paying such penalties.

Definition 8 *A party p may* reliably report *on event e if there is no SLA $(p, (c, o)) \in L$ or SLA $(c, (p, o)) \in L$ for any other party c such that $e \in \pi(o)$. The set of parties in a given scenario that may reliably report on an event e is denoted $\tau(e) \subseteq P$.*

Provided that a party may reliably report on an event, another party may choose to trust them:

Definition 9 *A party q may choose to trust a party p to report on an event e, if p can reliably report on e. For each party p and each event e, I define $\tau(p, e) \subseteq \tau(e)$ to be the set of participants that p* trusts to report *on the event e.*

Definition 10 *A party p may* monitor *an event e if $p \in \rho(e)$ or there exists a party $q \in \tau(p, e)$ who can monitor the event independently of p and who p trusts to monitor that event. For an arbitrary subset of all parties $M \subseteq P$ I recursively define a generic mapping from events to parties in M that can monitor the events, $\mu_M(e) = \{p \mid p \in M \wedge (p \in \rho(e) \vee \exists q.(q \in \tau(p, e) \wedge q \in \mu_{M-\{p\}}(e)))\}$. I therefore define $\mu(e) = \mu_P(e)$ as the set of all parties that can monitor an event e.*

Note that the set of parties who may be trusted depends on the set of SLAs issued. Therefore, in our scenario, C can only choose to trust I or S to report on events y and z (which C cannot monitor directly) if I or S do not offer or receive any SLAs related to these events.

Definition 11 *Given an SLA $(p, (c, o))$, a party q can* monitor *the SLA if it can monitor all events $e \in \pi(o)$.*

For example, to monitor an SLA related to the observation $y - x < t_1$ a party must be able to monitor both y and x, in order to determine the arrival and departure times of a request. If S issues $s_1 = (S, (C, z - x < t_1 + t_2))$ to C and I also offers $s_2 = (I, (C, w - z < t_3))$ to C, and no other SLAs are made, then s_1 will possible be monitorable to S, because S can directly observe z and might choose to trust I to report on x, because I offers no SLAs pertinent to x. C on the other hand cannot monitor s_1 because it cannot trust either S or I to report on z. Similarly s_2 is directly monitorable by I but cannot be monitored by C for the same reason as s_1.

Monitorability of an SLA is particularly desirable for a party that is the client or the provider of the SLA, as that party can know what penalties should be paid. In a fair scenario it would be desirable for both parties to be able to monitor the SLA, since then neither party could cheat the other without the other party being aware of it.

Definition 12 *An SLA, $(p, (c, o))$, is* mutually monitorable *if p and c can monitor the SLA.*

Supposing that S offered C, $s = (S, (C, z - y < t_2))$ and I and C offered nothing. s would be mutually monitorable if C trusted I to report on z and y.

Ideally an SLA would be monitorable by a third party, trusted by both client and provider to report honestly. Since the third party was trusted, it could be relied upon to arbitrate disputes between the client and provider.

Definition 13 *Given an SLA (p, (c, o)), if there exists a third party t such that for all $e \in \pi(o)$, $t \in \tau(c, e)$ and $t \in \tau(p, e)$ then I say the SLA is* arbitratable *by t, since both parties trust t to monitor the SLA.*

Supposing that S offered C, $s = (S, (C, y - x < t_1))$ and I and C offered nothing. s would be arbitratable by I providing that I was trusted by both C and S.

In the preceding example, and others above, there seems to be something intuitively wrong with the SLAs offered, in that parties attempt to insure the behaviour of actions that they do not themselves

perform. To avoid this I introduce a notion of guarantees, and subsequently characterise safe SLAs as being those that rely either on guarantees of which the provider is capable, or subordinate guarantees acquired from elsewhere.

Definition 14 *A party $g \in P$ may, by their actions, be able to guarantee that an observation holds.* Guarantees *are modelled like requirements as a pair $(g, o) \in G$. The guarantor must be able to monitor all events pertinent to the observation. The guarantor must also perform actions that cause a subset of the events pertinent to the observation, i.e. there must exist $e \in \pi(o)$ and $a \in A$ such that $g = \alpha(a) \wedge e \in \epsilon(a)$.*

Recalling that observations are predicates over the attributes of events, it is clear that to guarantee that an observation holds a guarantor must meet the two conditions defined in the observation. By causing some pertinent events, the guarantor can vary the values of the attributes of these events, thereby causing the observation to hold. However, to determine how this should be done, the party must also be able to determine the values of the attributes of the other pertinent events. In general, a party's capacity to guarantee observations may therefore depend on what events are monitorable to that party, and hence on the SLAs that are offered in a scenario.

In our example, the ISP and the service provider can guarantee several observations regardless of what SLAs are offered. The ISP can control the time taken to deliver the request to the server, once it has been received by the network. Note that the ISP performs the action that causes y and can always monitor x directly. The service provider controls the time taken to perform processing once the server has received the response. The ISP again controls the time taken to deliver the response once it has been received by the network. The following guarantees are therefore included in the model: $G = \{(I, y - x < t_1), (S, z - y < t_2), (I, w - z < t_3)\}$.

Definition 15 *An SLA $(p, (c, o))$ is* safe to issue *if p can guarantee o, i.e. if $(p, o) \in G$, or p can obtain an SLA $(q, (p, o))$ from a second party q, or if these conditions can be satisfied for all observations in any set of observations upon which o depends. I.e. for all $o' \in U$ where $U \in D(o)$, p can either guarantee o' or obtain an SLA for o' from a second party.*

If an SLA is safe to issue, the provider p may be liable to pay penalties when requirements are not met due to the actions of other parties, but will also receive penalties, which, appropriately negotiated, will obviate their risk. In the case of an SLA that is unsafe to issue, the provider may have to pay penalties due to the actions of other parties without receiving compensation themselves.

Having defined and motivated a set of characteristics for individual SLAs, I also describe systems of SLAs as follows:

Definition 16 *Let $G_p = \{o \mid (p, o) \in G\}$, $S_p = \{o \mid (q, (p, o)) \in S\}$ and $R_p = \{o \mid (p, r) \in R\}$, then the system is satisfactory to p iff $G_p \cup S_p \models R_p$. A system is* satisfactory *overall if it is satisfactory for all parties that it contains.*

In other words, a system of SLAs may be characterised from the point of view of a party contained in it as satisfactory if all requirements of the party are insured by a combination of SLAs offered to the

party and guarantees that the party provides themselves.

Definition 17 *A system of SLAs is* safe *from the point of view of a party if all SLAs that the party issues are safe to issue. A system of SLAs is* safe *overall if it is safe for all parties.*

Definition 18 *A system of SLAs is* monitorable *from the point of view of a party if all SLAs issued or received by the party are monitorable by the party. A system of SLAs is* monitorable *overall if it is monitorable by all parties.*

Definition 19 *A system of SLAs is* arbitratable *if all the SLAs it contains are arbitratable.*

Finally it may be desirable in an analysis to rule out the following types of system of SLAs:

Definition 20 *A system of SLAs S may also be characterised as* redundant *if there exists $S' \subset S$ and S' is both satisfactory and safe.*

Definition 21 *A system of SLAs may be characterised as* reciprocal *if it contains two SLAs $s_1 = (p, (c, o))$ and $s_2 = (c, (p, o))$. In other words, two parties exchange SLAs with respect to an observation.*

5.1.2 Monitorability analysis

A particular system of SLAs may be characterised as described in the previous section, depending on the degree of satisfaction, safety and monitorability afforded to its parties, and the possible redundancy or reciprocity of its SLAs.

Searches for systems of SLAs with particular characteristics are possible by keeping most of the model constant, then varying the set of SLAs used. For example, we might ask for a given scenario: what sets of SLAs are safe, satisfactory and monitorable?

To identify sets of SLAs possessing a specific set of the characteristics defined in the previous section, one could in principle generate all combinations of SLAs, classify each, and then accept or reject systems of SLAs according to their classification.

The maximum number of possible SLAs in a given scenario is $|O| \times |P| \times (|P| - 1)$.

The number of combinations of SLAs is therefore $2^{|O|\times|P|\times(|P|-1)}$. This is potentially a very large number, suggesting that a more intelligent strategy for identifying useful combinations of SLAs is needed.

Depth-first search is an appropriate technique for finding sets of SLAs. I propose the following algorithm for generating and testing sets of SLAs, presented in pseudo-code:

```
procedure DEPTH_FIRST()
begin
    return DEPTH_FIRST({}, {})
end

procedure DEPTH_FIRST(tentative, tried)
begin
    result := {}
    next := FILTER(tentative,
        GENERATE(tentative, tried))
```

```
    for each n in next
        result := result union
            DEPTH_FIRST({ n } union tentative, tried)
        tried := tried union { n }
    if ACCEPT(tentative) then
        result := result union { tentative }
    tried = tried minus next
    return result
end

procedure GENERATE(tentative, tried)
begin
    result := {}
    for each c in P
        for each p in P
            for each o in O
                if not c = p then
                    s := (c, (p, o))
                    if not s in tentative and
                        not s in tried then
                        result := result union { s }
end

procedure ACCEPT(tentative)
begin
    return true
end

procedure FILTER(tentative, next)
begin
    return next
end
```

The algorithm as presented will generate all possible combinations of SLAs. Potential efficiency benefits of the approach rely on redefining the heuristic operation `FILTER` to focus the search of the algorithm. `ACCEPT` may also be redefined to narrow the selection criteria for sets of SLAs, for example to accept only sets of SLAs with a minimum level of monitorability.

Note that the algorithm maintains a tentative set of SLAs that may be added to the result if they pass a test defined by `ACCEPT`. However, `GENERATE` first attempts to generate candidate SLAs to add to this set, these are filtered by `FILTER`, and then `DEPTH_FIRST` is recursively called to investigate each resulting tentative set. A set of SLAs that have already been tried is also maintained to avoid repeatedly trying to add the same SLAs in a different order.

A possible rewrite for `FILTER` uses the tentative set to identify requirements that are not yet satisfied if the SLA is to be safe and satisfactory, and eliminates or defers the consideration of SLAs that do not have the potential to contribute to the requirements. The dependency relationship can guide this. `FILTER` may also remove all SLAs from the extension set if `ACCEPT` will reject the tentative set and all supersets, as will be the case if `ACCEPT` rejects redundant or reciprocal sets.

Note that without any modification to `FILTER` the algorithm as presented is no more efficient than any other method for generating and testing all possible combinations of SLAs. In the worst case for a particular scenario and set of criteria, all systems of SLAs will meet the criteria. In general therefore no algorithm for this purpose can have complexity less than $\mathcal{O}(2^{|O|\times|P|^2})$.

Sat	Safe	Non-red	Non-rec	Non-client	Mon	Arb	Systems considered	Solutions
–	–	–	–	–	–	–	–	$\sim 6.9 \times 10^{10}$
✓	–	–	–	–	–	–	–	$\sim 6.6 \times 10^{10}$
–	–	–	✓	–	–	–	–	$\sim 3.9 \times 10^{8}$
–	–	–	–	✓	–	–	–	$\sim 1.7 \times 10^{7}$
✓	✓	✓	–	–	–	–	16001	281
✓	✓	✓	✓	–	–	–	7696	122
✓	✓	✓	✓	–	✓	–	7696	1
✓	✓	✓	✓	✓	–	–	3571	34
✓	✓	✓	✓	✓	✓	–	3571	1
✓	✓	✓	✓	✓	✓	✓	3571	0

Table 5.1: Results of a monitorability analysis for the ASP scenario, with performance of depth-first search algorithm

5.1.3 SLAs for the ASP scenario

The example scenario includes three parties and six observations. $2^{3\times2\times6} = 2^{36} \sim 6.9 \times 10^{10}$ distinct sets of SLAs are hence possible.

We are interested in the sets with the following properties: safety, satisfaction, non-redundancy, non-reciprocity, sets in which the client issues no SLAs (non-client), monitorability and arbitratability. We will also be interested in sets with combinations of these properties.

Some of these sets can be discovered with the depth-first search algorithm described in the previous sections. Others do not permit sufficient narrowing of the search space to render this approach feasible, but the number of these sets can be determined analytically.

I present here analytical solutions for the total number of satisfactory sets, the total number of non-reciprocal sets, and the total number of non-client sets.

The number of satisfactory sets can be determined by considering the SLAs that must be offered to the client to satisfy its requirement $w - x$. These must insure at minimum $w - x$ or any dependency set for $w - x$, which in this case may include any other observation. A total of 36 possible SLAs may be offered in this scenario. However, we are only interested in those that offer guarantees to the client, of which there are 12 (either I or S may offer an SLA for any observation to C). We can therefore determine the total number of satisfactory sets of SLAs by determining how many combinations of these 12 SLAs result in the satisfaction of the client's requirement, then multiplying this number by the number of combinations of the SLAs with which we are not concerned.

There are $2^{12} = 4096$ combinations of SLAs that may be offered to the client. A truth-table inspection of these combinations reveals 3927 to be satisfactory. The total number of satisfactory SLAs is equal to $2^{24} * 3927 \approx 6.6 \times 10^{10}$.

A Java implementation of the truth-table analysis for satisfaction is available online [114].

The number of non-reciprocal SLAs can be determined straightforwardly. Each possible SLA has a reciprocal SLA with which it should not appear, making independent 18 pairs, any one of which should not appear. For each pair, in any given system, neither SLAs may appear, either may, or both may, making a total of four possibilities, one of which is unacceptable. The outcomes for each pair are independent in a given system, hence there are $2^{36} * (3/4)^{18} = 3^{18} \sim 3.9 \times 10^{8}$ non-reciprocal sets of

SLAs.

One may wish to make the restriction that the client does not offer any SLA. This will reduce the space of the search to $2^{24} \sim 1.7 \times 10^7$

To obtain further results I employed a Java implementation of the depth first search algorithm to discover non-redundant sets of SLAs, and those with minimal levels of monitorability. Note that non-redundant sets of SLAs are by definition both safe and satisfactory.

The non-redundancy, non-reciprocity and non-client constraints all have the effect of considerably limiting the size of the search space when employing depth first search. This is because a redundant or reciprocal set, or a set containing a client SLA cannot be improved by the addition of SLAs.

A summary of the analytical results and the results of the search algorithm is shown in Table 5.1.3. The Java implementation is available under an open-source licence for inspection and modification [114].

The most significant result of this analysis is that in exactly one of these arrangements can all sets of SLAs be monitored by the parties to them. This is true whether or not we permit the client to offer SLAs.

In this scenario, for a system of SLAs to be safe and satisfactory both S and I must issue SLAs supported by guarantees contributing to C's requirements. Hence all parties will be financially involved in every contractual situation, and no party can be trusted to report any events that occur remote from another party. Therefore monitoring is only possible directly, and hence only SLAs between adjacent parties can be mutually monitored, namely, contracts between C and I, and I and S. Only one scenario meets this requirement. It consists of the contracts $(I, (C, w - x < t))$ and $(S, (I, z - y < t_2))$. The ISP guarantees that the service will perform correctly across its interface with the client. It is capable of guaranteeing that the request reaches the server in a timely fashion, and that any response makes it back in time. To fully guarantee the round-trip time of the service the ISP must only obtain a guarantee from the service provider that the service will complete in good time.

That no arrangement can be arbitrated is obvious without applying the search algorithm. Because all parties in the scenario must be involved in contracts to satisfy C's requirement, no financially independent third party can be present to observe any interaction.

That the scenario is only monitorable in one system of SLAs is a highly significant result. This system of contracts requires the ISP to offer guarantees on the received quality of an electronic service at the interface to the client, effectively forcing the ISP to act as a re-seller of application services, a business model not adhered to in practice today. Service constraints will be required at both the interface to the client and the interface to the service. The guarantees required in both places will be of the same form, although the constraints at the service interface will need to be tighter to accommodate any delay in the network whilst guaranteeing requirements at the client interface. Therefore to achieve monitorable end-to-end QoS guarantees for ASP, only one type of SLA language need be used. There is no need for a separate language to describe network QoS, for example. However, ISPs will have to offer ASP SLAs.

The 'systems considered' column in Table 5.1.3 demonstrates of the efficiency of the depth-first search algorithm configured with the indicated heuristics. Directing search towards safe, satisfactory

and non-redundant sets, and pruning sets that fail to meet these requirements reduces the search space from $\sim 6.9 \times 10^{10}$ combinations to 16001 combinations. Non-reciprocal and non-client constraints further reduce the search space.

5.1.4 Multiple ISPs

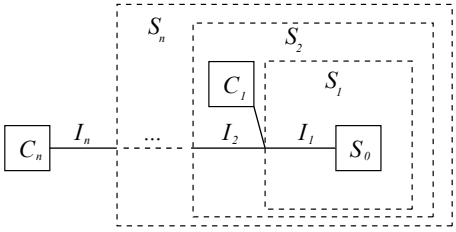


Figure 5.2: Monitorability is possible for ASP SLAs across chains of ISPs by regarding ISPs encapsulating the service as service providers, hence $I_i = S_i$ for $i > 0$. Clients may be embedded in any network

The results of the previous section may be generalised to scenarios including multiple ISPs and clients distributed in the network. Clearly a sequence of SLAs can be established between each client and the service, such that each SLA is made between two adjacent parties, one serving as the client and the other as the service provider. This situation is shown in Figure 5.2. In the figure the original service S_0 is embedded in a network I_1. I_1 can hence provide the service to C_1. It can also exchange requests and responses with the linked network I_2. Multiple networks are linked to provide a path to client C_n.

Clearly a system of SLAs that is monitorable can be provided for C_1, as this is the same situation as analysed in the previous section. That a system can also be provided for clients embedded in any network I_i, where $i = 2 \ldots n$ can be seen by considering the fact that in exchanging requests and responses with I_2, I_1 is behaving exactly like a service embedded in network I_2. This is indicated using the dashed box which re-identifies I_1 and S_0 as S_1. Therefore provided I_1 and I_2 make an SLA insuring the timeliness of responses at their mutual interface, then it will also be possible for I_2 to offer such an SLA to any client embedded in their network. Clearly the SLA between I_1 and I_2 will be mutually monitorable, since it need only concern events occurring at the parties' mutual interface. This argument applies inductively for any number of network links, so it will be possible to provide a chain of monitorable SLAs to insure the timeliness requirements of client C_n. In general, clients may be embedded in any network provided a path to the original service exists, and SLAs are can be made at each network boundary.

This analysis establishes that when multiple ISPs are involved in delivering the service, then systems of SLAs exist that are at least mutually monitorable. What has not yet been established is whether in this scenario arbitratability can be achieved for some or all parties.

This discussion also reveals that mutually-monitorable systems of SLAs for ASP may need to consist of large numbers of individual SLAs, specific to a given client/server relationship, and that network-service providers will need to be involved in the negotiation of all of these. Given that network-service providers do not currently engage in this kind of business, this result surely has one of the following implications: the need for monitorability of SLAs may (continue to) be ignored, which would be unfortunate given the importance of monitorability when producing protectable SLAs; ISPs may consider

engaging in this kind of business in the future, implying the need for research into reducing the costs implied by doing so; or a technological solutions may need to be sought to allow parties to obtain reliable measurements from locations in networks that they normally could not directly observe. This final approach, which may be called the design of trusted-monitoring platforms, might permit the measurement of end-to-end properties, allowing SLAs with network providers to be made at the network-level of abstraction. However, whether such monitoring is possible in practice remains the subject of future investigation.

5.2 Approximate monitorability

Fair administration of SLAs requires accurate data concerning the behaviour of services. In at least one important case this data must be provided by a party that is not inherently trustworthy, but whose reporting can be monitored. This has the potential to occur whenever an SLA is mutually-monitorable but not arbitratable, in which case either the client or the provider of the service has the potential to gather a log of service behaviour, but (by the assumption of my monitorability model) neither party can be trusted by the other.

The calculation of violations of an SLA must always be made in relation to some account of service behaviour. Depending on their agreement, the client or the provider in a mutually-monitorable relationship may be satisfied with their peer calculating the penalties required based on their own records of service behaviour, and paying or demanding payment of penalties as appropriate. However, even if a party is prepared to allow their peer this responsibility under normal circumstances, they would be foolish if they did not also monitor the service themselves to some degree, in order to check the honesty of their peer. This raises the possibility that the peer may be found to be cheating, or calculating penalties based on faulty monitoring data.

Under these circumstances, one of two things can happen. The injured party can either break-off the agreement, or the parties can come together to agree a reconciled account of service behaviour from which to calculate revised penalties.

If the injured party chooses to break off the agreement, they will need to have a basis for asserting that the other party has violated the agreement, or risk being penalised themselves for breaking a contract with value to their peer. They could simply claim that the other party had calculated penalties incorrectly, either maliciously or due to incorrect monitoring. To do so, the SLA would need to include some constraint that the penalties calculated should be somehow related to the true behaviour of the service. However, the calculating party must always first measure the behaviour of the service, and measurements of any physical quantity always contain a component of error. Therefore, if penalties were naïvely defined in terms of the true behaviour of a service, this would potentially provide opportunities for a party to spuriously contest penalties on the basis of small amounts of error in the measurements on which it was calculated. This would reduce the usefulness of the SLA in mitigating risks for both parties. Therefore, an SLA that is both fair and useful should include a constraint on the measurements from which penalties are calculated which limits inaccuracies without excluding them altogether.

Similarly, an accuracy constraint is needed on the presented logs in the case where the client and

provider of the service must agree a reconciled account of service performance. This is because the parties have no third-party upon whom to rely, and neither party can fully establish the validity of their own account of the service performance by technical means alone. In principle, the agreed account need bear no resemblance to the real behaviour of the service, providing the parties agree. However, in the event of a disagreement between the parties, honest parties who are concerned with the real behaviour of the system will wish to be able to argue that their counterpart has neglected their responsibilities. It is therefore necessary to include in such SLAs an obligation that parties report the behaviour of the service accurately.

The inevitable presence of error in any measurement of a physical system raises two problems in the face of these requirements for accuracy constraints. First, how can such an obligation be formulated to permit a tolerable degree of error in reporting, at the same time penalising higher levels of error, whilst respecting the right of the client to vary their utilisation of the service, and both parties to conceal details of their monitoring solutions. Second, how can conformance to such a constraint be checked, given that the true performance of the system cannot be determined with total certainty.

In the following sections I formulate such a constraint and explain how a statistical hypothesis test can be used to tell with some degree of confidence whether the constraint has been violated. Since complete confidence in this result is not possible, we introduce the term 'approximately monitorable' to refer to this type of constraint.

While describing the constraint I also give advice as to how parameter values may be chosen so that the parties may be confident that they can meet their obligations, assuming that they understand the error characteristics of their monitoring process.

The constraint chosen requires the parties to provide minimal information concerning the error characteristics, and by implication the implementation of their monitoring solutions. I also investigate the degree of fraud that is possible assuming that a party has perfect knowledge of their error characteristics, but need not reveal this information.

5.2.1 Accuracy constraint

The constraint that I have chosen is designed to detect when a set of reported measurements is statistically unlikely to be an accurate account of true behaviour. Violation of the constraint would enable the injured party to seek redress for misreporting of service behaviour by the other party.

A log consists of a sequence of measurements X_i of event values μ_i where $i = 1, \ldots, n$. Our constraint therefore requires of the log that:

$$pr(|X_i - \mu_i| > e) \leq 1 - c \tag{5.1}$$

Where e is the specified error margin, and c a specified confidence in the measurement. μ_i, the true value of an event attribute, is always unknown.

Since a party is only required to comply with the accuracy constraint, I assume the probability p of a measurement in a given log being accurate is:

$$p = pr(|X_i - \mu_i| > e) = 1 - c \tag{5.2}$$

A measurement is *erroneous* if it falls outside of the error interval centred on the true value. I wish to limit the number of erroneous values in any given log. Although the true value μ_i of a measurement can never be known, it can and must be referred to in our accuracy constraint as the basis for defining accuracy.

Let d denote the number of erroneous measurements in a particular log. I wish to prohibit the reporting of logs containing improbably large numbers of erroneous values.

Assuming that a party reports honestly, using a monitoring system with the above error characteristics, the probability of a log of size n containing d erroneous measurements is given by the binomial distribution:

$$pr(d) = \binom{n}{d} p^d (1-p)^{n-d} \tag{5.3}$$

We wish to bound the likelihood that the log contains more than a certain proportion of erroneous values. An additional parameter, α, is therefore specified in the SLA. For a particular size of log, n, and choice of SLA parameters e and c, we can therefore determine an upper bound on d, d_0, which is the greatest integer such that:

$$\sum_{d=d_0+1}^{n} \binom{n}{d} p^d (1-p)^{n-d} < \alpha \tag{5.4}$$

In other words, provided the measuring party is respecting the constraint, the likelihood that the log contains more than d_0 errors is less than α.

The formulation of the constraint is analogous to a statistical hypothesis test, in which d is the test statistic, and the null hypothesis is that the log is honest. α is equivalent to the type I error rate for the test – the probability that an honest log is rejected as dishonest by the constraint.

5.2.2 Approximate monitorability of the accuracy constraint

It is not possible to determine with certainty that a party has conformed to the constraint described in the previous section, because determining the number of erroneous values d requires the true values of the events μ_i to be known, and they cannot be known with certainty.

However, if a contract is monitorable then all parties to the contract will be able to obtain trusted measurements of all events pertinent to the contract. Hence, for a party p, it will be possible to approximately monitor the conformance of an untrusted party, q to the accuracy constraint by comparing the untrusted log produced by q, measurements X_i, with a trusted log, measurements Y_i, produced by p (possibly with the help of third parties trusted by p). I assume that the logs are that same size and that events in the two logs can be correlated. This assumption is likely to be valid in a mutually-monitorable situation.

The approximate monitoring of the accuracy constraint is achieved via a statistical hypothesis test. The null hypothesis is that the untrusted party has produced an honest log. The alternative hypothesis is that untrusted party is cheating, or otherwise failed to conform to the accuracy constraint.

H_0: Contractor is honest

H_1: Contractor is unable or unwilling to conform to the accuracy constraint.

We wish to detect erroneous values. It is not possible to compare X_i to μ_i, but we can compare it to Y_i. Y_i is trusted, and we assume that some characteristics of its error distribution are well understood. For now suppose that the trusted log is at least as accurate as required by the accuracy constraint. Therefore, if the absolute difference between the two logs is greater than $2e$ then the probability that X_i lies further than e from the true value μ_i is related to the confidence we have that Y_i lies within the error interval of the true value, which in the worst case will be given by c. Note that if Y_i is within e of the true value, and X_i is greater than $2e$ of Y_i then X_i cannot also be within e of the true value. I shall therefore provisionally adopt d', the number of cases where $|X_i - Y_i| > 2e$, as the test statistic for our hypothesis test.

Large values of d' will favour H_1. Similar to the formulation of the constraint, it is therefore a matter of comparing this statistic to a threshold value d'_0 such that:

$$pr(d' > d'_0 \mid H_0\ true) = \alpha \tag{5.5}$$

In order to determine d'_0 we need the sampling distribution for d' when H_0 is true. This is again given by the binomial distribution:

Let:

$$p' = pr(|X_i - Y_i| > 2e \mid H_0\ true) \tag{5.6}$$

The sampling distribution of d' is hence:

$$pr(d') = \binom{n}{d'} {p'}^{d'} (1-p')^{n-d'} \tag{5.7}$$

The problem, therefore, is to find an expression for p' in terms of e and c. This could be determined exactly if the error distribution for each party were known, but in practice we can only assume (under the null hypothesis) that the distributions of the parties conform to the parameters given in the accuracy constraints. The best that is therefore possible is an upper bound for p'.

The Chebychev inequality [139] states that given any random variable X where $E(X) = \mu$ and $Var(X) = \sigma^2$ then:

$$pr(|X - \mu| > k\sigma) \leq \frac{1}{k^2} \tag{5.8}$$

Therefore, under the null hypothesis:

$$pr(|X_i - \mu_i| > k\sigma) \leq \frac{1}{k^2} \tag{5.9}$$

The untrusted party has agreed that the error in their log will obey the rule

$$pr(|X_i - \mu_i| > e) \leq 1 - c \tag{5.10}$$

Because we can assume under the null hypothesis that the untrusted party is honest, but no better, to obtain our upper bound for p', we must assume:

$$pr(|X_i - \mu_i| > e) = 1 - c \tag{5.11}$$

Equating the parameters of the constraint with those of the Chebychev inequality, we obtain:

$$k\sigma = e \tag{5.12}$$

And:

$$1 - c = \frac{1}{k^2} \tag{5.13}$$

Resulting in the following worst-case relationship between the standard deviation of the error distributions and the parameters of the constraint:

$$\sigma = e\sqrt{1 - c} \tag{5.14}$$

Or:

$$e = \frac{\sigma}{\sqrt{1 - c}} \tag{5.15}$$

Now consider $X_i - Y_i$. I make the following assumptions:

1. $E(X_i - \mu_i) = E(Y_i - \mu_i) = 0$
2. The variance of the trusted log, σ_Y^2 is known.

The first assumption, that the measurements are unbiased, is reasonable, because any systematic bias on the part of either party will easily be detected when the two logs are compared, and can either be easily rectified or will swiftly result in a breakdown of relations between the two parties. Occasionally both parties may suffer from similar biases, which may hence be overlooked. This will not be problematic from the point of view of obtaining an agreement between the parties, and may be rectified if detected later.

The second assumption reflects the fact that the trusted log has been obtained via a measurement process the error characteristics of which are known to some degree. I have already assumed that the measurement process gives results conforming to the accuracy constraint on the untrusted log. Therefore if nothing else is known I may assume $\sigma_Y^2 = \sigma$.

Since in the worst case according to the null hypothesis the variance of the untrusted log is σ then the variance of $X_i - Y_i$ will be:

$$Var(X_i - Y_i) = \sigma^2 + \sigma_Y^2 \tag{5.16}$$

By introducing a constant r such that:

$$\sigma^2 + \sigma_Y^2 = r\sigma^2 \tag{5.17}$$

I state my assumptions in the form:

$$E(X_i - Y_i) = 0, \quad Var(X_i - Y_i) = r\sigma^2 \tag{5.18}$$

Therefore, by Chebychev again

$$pr(|X_i - Y_i| > k\sqrt{r}\sigma) = \frac{1}{k^2} \tag{5.19}$$

The original relationship still holds:

$$p' = pr(|X_i - Y_i| > 2e) = pr\left(|X_i - Y_i| > 2\left(\frac{\sigma}{\sqrt{1-c}}\right)\right) \tag{5.20}$$

Equating the parameters, and cancelling the σ terms:

$$k\sqrt{r} = \frac{2}{\sqrt{1-c}} \tag{5.21}$$

$$k = \frac{2}{\sqrt{r}\sqrt{1-c}} = \frac{2}{\sqrt{r-rc}} \tag{5.22}$$

And:

$$p' = pr(|X_i - Y_i| > 2e) = \frac{1}{k^2} = \frac{1}{\left(\frac{2}{\sqrt{r-rc}}\right)^2} = \frac{r-rc}{4} \tag{5.23}$$

d'_0 can hence be determined for given values of n and c by substituting $p' = \frac{r-rc}{4}$ into the binomial distribution.

This is adequate to demonstrate that the accuracy constraint as specified in Section 5.2.1 is approximately monitorable. A further generalisation of the hypothesis test can be obtained by observing that interval by which measurements in X and Y must differ to contribute a fault to d' simply introduces an arbitrary constant into the above expression for p'. I originally assumed the interval to be $2e$. If I instead assume it to be te where t is some constant, then the expression for p' becomes:

$$p' = pr(|X_i - Y_i| > te) = \frac{1}{k^2} = \frac{1}{\left(\frac{t}{\sqrt{r-rc}}\right)^2} = \frac{r-rc}{t^2} \tag{5.24}$$

Of course d' must be recalculated based on the value of t chosen. If we choose a difference of 1, and assume that the trusted log meets the accuracy constraint and no better, giving $r = 1$ then p' is given by:

$$p' = 1 - c \tag{5.25}$$

Which is the same as the expression for p in the accuracy constraint. Therefore, comparing a conforming log with an un-trusted log as if the conforming log represented the true behaviour of the service produces false-positives at a rate acceptable to the constraint.

5.2.3 Choosing parameter values

Assuming that a party only knows the standard deviation of their error process, and wishes to guarantee that they measure honestly, the agreed value of e will be related to the agreed value of c by:

$$e = \frac{\sigma}{\sqrt{1-c}} \tag{5.26}$$

Hence if a confidence of $c = 0.99$ is required then a value for e of 10σ is required. This is highly conservative.

If a party understands more details of their error distribution they may be able to accept a tighter bound on e.

A party may negotiate values of e and c such that the true probability p_T of an erroneous value is less than the Chebychev bound. In this case the party will be able to insert purposeful erroneous values in proportion $p - p_T$ to the total size of the log n. Note that the client, through its ability to issue or withhold service requests controls the size of the log.

Given the choice of SLA parameters, this behaviour is impossible to prevent. However, negotiating tighter bounds for e and c will reduce the degree of cheating possible regardless of the true distribution of error in the measurement process of either party. The accuracy of measurement guaranteed may therefore be regarded as a discriminating point in a competitive market of services governed by SLAs.

For example, for a measurement confidence $c = 0.95$ and a type I error rate of $\alpha = 0.05$, in a log of size $n = 1000$ we would tolerate up to $d'_0 = 33$ differences of greater than $2e$ between the party's log and another trusted log with unspecified distribution conforming to the accuracy constraint. If the measurement regime of the untrusted party is in fact perfect with $pr(|X_i - \mu_i| > e) = 0$, then the party may be confident in introducing up to 33 purposeful errors into its log prior to reconciliation between the parties.

5.3 Related work

As discussed in detail in Chapter 8 a number of current and prior efforts to design languages for ASP SLAs or service offerings have been proposed, most notably WS-Agreement [100], WSLA [34] and WSOL [132]. To the best of my knowledge, no previous language explicitly addresses the need for monitorability, or provides any constraints on or discussion of measurement error either in the design of the language or any of its related documentation. However, all of the languages cited here require or permit extension to define metrics, so they have the potential to address these issues.

The management of error in performance measurement for analysis and benchmarking is an important related topic which has been well covered by prior work [40].

Concepts related to monitorability have been touched upon by previous work in the area of policy management. It is conventional in policy languages to define rights and obligations that may attach to managers [122]. These rights and obligations are scoped according to *management domains* containing policy objects, where a manager 'sees' one or more management domains. Assuming management domains are correctly modelled, the monitorability of policy objects from the point of view of managers who must execute policy may be ensured at parse time in languages such as Ponder [16].

The execution of an obligation policy is usually the responsibility of the manager to whom it applies. It may be that the manager is not trusted so there is a requirement to check whether an obligation has been fulfilled. [11] provides an algorithm for determining whether obligations conforming to a given model have been fulfilled, but assumes that all relevant events are visible. [12] provides a logic whereby accountable managers may prove that an obligation has been fulfilled with reference to a log of actions. A notion of observability of actions is introduced to constrain the model. However, accountability relies on a universally trusted logging system being available within the manager's domain. It is not clear how this could be implemented, or if it could be used to monitor negative obligations. To the best of our knowledge, no work has yet been done on checking policies for monitorability under conservative trust assumptions similar to those adopted in this paper.

A highly influential paper with apparent relation to our work on accuracy is Lamport's clock synchronisation paper [50]. The accuracy constraint discussed in the preceding sections does not require parties to synchronise clocks in order to measure the timing of mutually observable events. Instead the constraint requires a measurement to be accurate with respect to the true time, and the means by which the parties should achieve this is not relevant.

Lamport described an algorithm with a bounded error for synchronising distributed clocks. This may potentially be helpful in situations such as our example scenario where the parties to the SLA are technically adjacent and so could share synchronisation information, and where the constraints are primarily concerned with relative rather than absolute timings. Measuring the time of events with high global accuracy is difficult, and may require specialist equipment, for example a GPS receiver, so a synchronisation based approach may be preferable. However, it will be necessary to determine how this interacts with trust assumptions I have made. For example, synchronisation protocols should not be employed that allow one party to maliciously alter the clock of another with whom they have an SLA.

5.4 Summary

In this chapter I have presented three significant contributions to the theory underlying SLAs, originally presented in [120]:

First, I presented and motivated a model and analysis technique for reasoning about the monitorability of systems of SLAs.

Second, I instantiated that model to perform an analysis on the important example of client/server computation taking place across a network owned by one or more third parties. In the case that the network is owned by a single provider, and trusted monitoring is not provided using any technical solution (such as tamper-proof monitors), I demonstrated for latency constraints that the highest level of monitorability possible is for all SLAs to be mutually monitorable, and I have described the single configuration in which this holds, namely that the ISP offers the client an SLA at the client's interface to the network, and the service provider offers an SLA to the ISP at the service provider's interface to the network. This is an extremely significant result as it implies that if the insurance of end-to-end QoS is to be offered using legalistic SLAs in the Internet then a major change will be required to the business model that ISPs currently operate. Alternatively, if this is not possible, it might suggest the criticality of future

research into embedding trusted monitoring solutions into network and service-provision infrastructure.

Finally, I considered support for SLAs that are mutually monitorable but no better. In this case, parties must agree on the behaviour of a service prior to determining the penalties to be paid in relation to a particular SLA. I observed that a naive constraint that the parties report service behaviour honestly would be impractical due to the inevitable presence of error in any measurement, but that without such a constraint an honest party would have no recourse should an agreement fail to be met. Allan Skene and I therefore designed a reporting constraint that described a limit on the number of errors a log could contain based on its size, the stated confidence the parties have in the measurements, and an agreed type I error rate for the test. I showed that this constraint could be approximately monitored by using a statistical hypothesis test to compare a log of measurements under test to a second log with known error characteristics. Naturally, approximate monitorability fails to detect a degree of cheating, which I quantified.

The material in this chapter provides a certain amount of theoretical machinery supporting the definition of an abstract extensible DSL for ASP SLAs. I showed that in the common case that timeliness constraints are desired by a client, and the scenario includes three parties, the client, the service provider, and the network service provider, then mutual monitorability was the best that could be expected from SLAs. However, this has the advantage that an SLA language need only describe the behaviour of electronic services and real-world behaviour, and not that of the network. Also, I presented an approximately monitorable accuracy constraint appropriate to this situation. In the next chapter I describe the design of a core SLA language capable of mitigating the risks in an ASP situation well. It achieves this by including support for constraints that are mutually-monitorable, and by incorporating the accuracy constraint described here in semantics for the calculation of violations and reconciliation procedures. Timeliness constraints have been the focus for discussion in this chapter. In the next chapter I show how useful throughput and reliability constraints can also be specified without compromising my key monitorability result for ASP scenario, and discuss how monitorability should be considered when designing constraints on real-world service behaviour.

Chapter 6

The SLAng language

In this chapter I describe the design of SLAng, an abstract, extensible domain-specific language for SLAs for ASP. SLAng is defined using a language specification written in the EMOF/OCL input format of the UCL MDA tools, consisting of a combination of EMOF structure, OCL constraints and natural language commentary, following the method which I described in Chapters 3 and 4.

SLAng incorporates abstract syntactic structures and accompanying semantics supporting the definition of timeliness, reliability and throughput conditions related to the behaviour of electronic services. Such conditions were identified in Chapter 2 as being required to allow the parties to mitigate fundamental risks inherent in the ASP provisioning scenario.

As described in Chapter 5, if we assume that the parties in a service-provisioning scenario will prefer to enter into mutually-monitorable SLAs in a safe manner, it is sufficient to provide vocabulary for these conditions as they apply at a single point in a network, rather than having to describe conditions on end-to-end service behaviour. In the support it provides, SLAng adopts this assumption, although, being an abstract language, does not preclude the addition of constraints of a different type in future revisions or extensions.

Since SLAng provides support for conditions that are mutually-monitorable, it defines the calculation of violations of the conditions based on evidence that is gathered according to the approximately-monitorable accuracy constraint described in Chapter 5, and also provides support for specifying when these calculations should be made, and how reconciliation between the parties may be initiated if needed.

SLAng has been under development for several years. In this chapter I first provide a brief history of the development of SLAng, in order to explain discrepancies between the description of SLAng given here and descriptions of the language presented in previous work, and also to explain further the thought process that led to the theoretical advances described in previous chapters.

Following that I describe the language in detail with reference to elements of the language specification. The language specification is written in a textual syntax, but in this section I use UML class diagrams to describe the structure of the language, according to standard practice when presenting EMOF models. An extended version of the language specification, supporting the definition of the SLAs elaborated in the next chapter, constitutes Appendix E. This appendix was typeset automatically from the language specification and extension elements using the LaTeX version of the `EMOFOCLDoc` tool developed as part of the UCL MDA tools. Example SLAs, presented in HUTN format, constitute Appendices C

and D.

6.1 The history of SLAng

SLAng was first described in [49], and was the product of research and development conducted primarily by Domenico Davide Lamanna. The motivation behind this work came from the observation that no SLA language has found broad adoption for use in the ASP domain, despite a variety of SLA languages having been previously described, and despite the strong need to manage Quality-of-Service (QoS) requirements in that domain.

The QoS delivered by an application service depends not only on its own implementation, but on other application services to which it subcontracts part of its functionality, and upon the quality of infrastructure services such as Internet Service Provision (ISP) and component hosting. It was perceived that although both general-purpose QoS description languages, and languages specific to relevant types of services, such as web-services, had already been defined, no single language provided adequate support for defining SLAs for the full range of services required to compose an application service. Also, that more parties than merely the overall provider and ultimate client of an application-service were involved in the ASP scenario. Infrastructure providing parties, such as ISPs and component hosting services may also be involved, and existing languages seemed to have no mechanism to accommodate this.

The original scope of SLAng was hence based on a model of a traditional N-tiered application service architecture, shown in Figure 6.1.

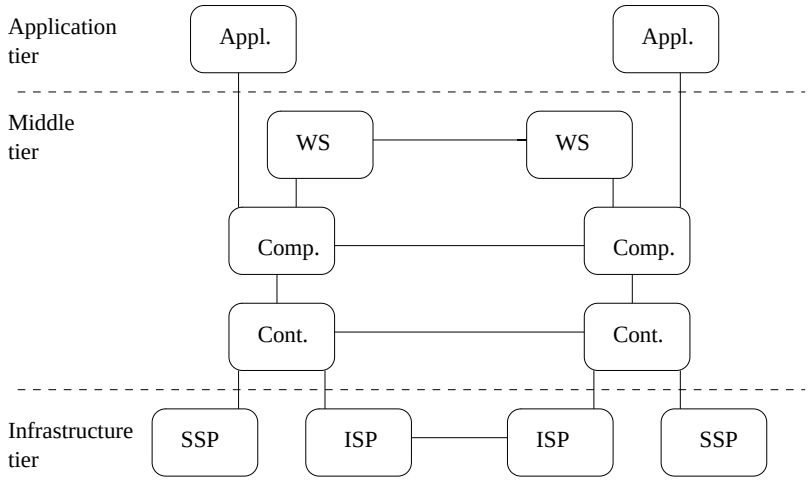


Figure 6.1: Service provision in three-tiered architectures

Architectural components are depicted as nodes in the model. These include client-facing applications, web-services, business logic components, containers providing runtime-support for these components, underlying networks services, and back-end storage (usually in the form of databases).

Arcs represent service-provisioning, potentially governed by SLAs. SLAng originally provided special syntax for each of the arcs in the model, defining QoS targets based on the type of service being provided. In an attempt to discover commonality between these different types of SLAs, we informally categorised them as *horizontal*, in which the client subcontracts part of its functionality to a service of the same type, or *vertical* in which infrastructure is provided to a client, allowing them to deploy a higher

level service.

The horizontal SLAs are: *Electronic-service* – between web services, or component services (the top two horizontal arcs in the figure), for the distribution of functionality; *Container* – between containers, for replication and load balancing; and *Networking* – between networks, for the sharing of network traffic. The vertical SLAs are: *Hosting* – between components and containers; *Persistence* – between a container and storage service provider; and *Communication* – between containers and Internet service providers.

Note that this formulation contains some inconsistencies. Interaction between applications, which are typically client programs, and the services underlying them is shown as a vertical, as the services represent infrastructure upon which the application depend. However, the nature of these interactions is identical to the horizontal electronic-service case. Also, in practice, application providers may need to purchase network services, an interaction not shown in the diagram. Also, no consideration is given to the real-world behaviour of application services.

The particular QoS targets (e.g. latency, reliability, throughput etc.) specified in each type of SLA were chosen based on a review of SLAs used in industrial settings, and recommendations provided to us by our industrial partner in the TAPAS project.

The first version of SLAng used an XML schema to define its syntax. Its semantics were described using natural language in a specification document. Some syntax and definitions, such as the definition of units, are reused in several SLA types.

The first version of the language was deemed unsatisfactory, and in retrospect this was because it failed to meet several of the requirements set out in Section 2.8. The definition of the language was highly imprecise and open to interpretation, which made it an unsound basis for specifying agreements. Lamanna ceased development of the language following its first version, and I took over its development, eventually leading to the development of the theoretical advances described in previous chapters of this dissertation, and to the redevelopment of the SLAng language.

The first major change to the language was as a result of adopting the model-denotational approach to its specification. The original ambition of the TAPAS project was that an SLA language be developed with a semantics using a denotational mapping to a stochastic process algebra, such as PEPA [33] or TIPP [32]. However, I objected to this, for the reasons discussed in Section 3.3.2. Consequently I proposed and implemented a model-denotational semantic for the language as an alternative. This also meant that the language was now defined using a meta-modelling formalism rather than an XML schema.

When specifying initial model-denotational semantics for the language, I confined my work to the syntax provided for electronic services, as the constraints required were less complex and better understood than those for hosting, network-service provision, or the other types of service that we anticipated needing to support. This was the work presented in [119].

Subsequently I began to consider implementing support for services of different types. When considering constraints over network-service and hosting provisioning, I became concerned that however precise an SLA was, it would be useless for a party if they could not tell if it were being respected. This

is very much a risk for the client in hosting provision where the service provider controls most aspects of the relationship, even to the extent of simulating the behaviour contributed by the client by executing the hosted component. This concern eventually led to the monitorability analysis technique presented in Chapter 5. I again applied this to the high-level example of electronic services first. In the absence of trusted-monitoring solutions, and assuming that mutually-monitorable SLAs are desirable, the results of my analysis indicate that there is no need to represent conditions relating to the behaviour of the network alone, and trusted monitoring would also clearly be required to monitor hosting relationships. I therefore focussed the design of SLAng on the specification of conditions related to electronic services.

The most recent modification to the approach taken with SLAng has been the decision to make the language abstract and extensible. In my initial review of alternative languages for ASP SLAs, I observed that many of the languages supported or required extension, often greatly at the expense of power and adequacy. I initially believed this to be due to a failure of the authors to correctly understand the role of SLAs in mitigating risks in the service-provisioning scenario, a judgement also supported by the observation that few of the languages discuss penalties. I felt that this assumption, combined with the assumption of a simple synchronous model of interactions between services, allowed the anticipation of all of the types of constraints that would be needed for an electronic service SLA.

This turned out to be wrong. When I began to attempt to evaluate a version of my language specified in this manner in real contexts, I repeatedly found situations in which my assumptions were violated. The basic assumptions concerning risk were fine, and latency, reliability and throughput were the constraints required for electronic services, but the services would be asynchronous, the parties would really care most about the real-world behaviour of the service, a service-credit system would be desired instead of escalating penalties, or the parameters of a latency constraint would need to vary depending on the state of an external process. Constraints on asynchronous services could be dealt with by a one-time extension of SLAng. However, extensions for other variations could not be said to capture anything particularly profound about ASP SLAs in general, so only served to highlight the inadequacies of the core language. Consequently, the problem was to find a way to balance the requirement for a powerful language with the need to support general expressiveness, and the approach I chose is described in Chapter 3. I also devised the metrics for power, adequacy and specificity described in Section 4.5 in order to assist in assessing the degree to which this has been successfully achieved.

Defining the language as abstract and extensible also plausibly admits the possibility of defining conditions related to real-world behaviour in SLAs (via extensions to the language specification). The consequences of all of these developments are that SLAng is now an abstract, extensible language, defined in EMOF, OCL, and English, with precise semantics, supporting the definition of mutually-monitorable SLAs for ASP, in particular conditions related to electronic services. It also bears no significant resemblance to the initial version of the language published in [49].

6.2 The SLAng language specification

In this and subsequent sections I describe the most recent version of the SLAng language specification. This specification is documented as part of the extended language specification in Appendix E. It is also

available under an open-source licence online [121].

Since SLAng follows a model-denotational approach the specification contains both syntactic elements and semantic elements. For ease of management, these are separated into parallel package hierarchies called `slang` and `services` respectively. Within each hierarchy, elements that are generic, and expected to be useful for defining any type of SLA are defined in the root. Elements specific to a particular type of service are included in sub-packages. At present SLAng only provides support for electronic-services, although this may change in the future. Figure 6.2 shows the package structure of the specification. Packages are only used to manage the specification and improve its readability. They have no effect on the meaning of the language.

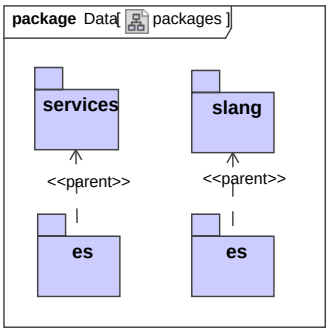


Figure 6.2: The package structure of the SLAng language specification

In the following sections I present class diagrams illustrating the structure of the syntactic and domain models making up the SLAng language specification. Both syntactic classes, which represent parts of an SLA document, and semantic classes, which represent services, parties and events in the ASP domain, may appear in these diagrams. I have adopted the convention of shading the syntactic classes lightly, and the semantic classes more darkly.

6.3 SLAs, parties and services

Figure 6.3 shows part of the generic structure of a SLAng SLA (that is, independent of the type of the service being constrained). SLAs define at least two parties, and some services. Service definitions are abstract since a more concrete service type must be specified. However, all services identify unique client and provider parties. Uniqueness is enforced using an invariant defined on `ServiceDefinition`. The semantics of a party definition is also shown. Each `PartyDefinition` identifies a unique party in the real world (uniqueness enforced by an invariant again). This is a straightforward denotational relationship.

The `SLA` class is concrete in the current version of SLAng. However, it has components, similar to `ServiceDefinition`, that are both mandatory, and abstract (indicated by an italic class-name, in the diagrams), and, unlike `ServiceDefinition`, have no more-concrete types. Therefore a complete SLA cannot be specified without extension to the language. Other properties of the `SLA` class are described in subsequent sections.

The `SLA` class includes a `uRI` attribute, which is to allow the author to identify the location where

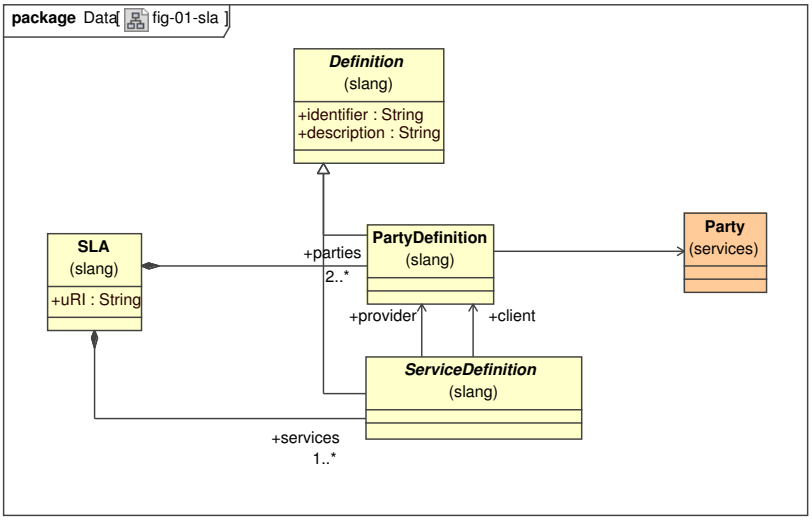


Figure 6.3: Party and service definitions in SLAng

a resource representing the definitive statement of the SLA resides. This is important, because a single SLAng SLA may have several concrete representations, for example both HUTN and an XMI representations. There is also the potential to develop software to reformat SLAng SLAs into even more human-friendly documents for the purpose of aiding comprehension of the SLAs. However, these types of transformations of an SLA document have the potential to introduce errors which in the worst case may modify the interpretation of the SLA. Identifying the definitive form of the agreement using a URI, which must be unique, reduces this possibility. Any interpreter of a concrete representation of a SLAng SLA should, according to my recommendations in Chapter 4, be able to follow unambiguous references to access a definitive version of the SLAng language specification. It will therefore be clear that this attribute should be present, and when interpreting an SLA this attribute can be checked to ensure that the definitive version is being used.

Naturally, it does not always make sense to make SLAs publicly accessible. However, URIs can also address secured resources.

6.4 Failures and violations

Two types of bad behaviour are possible in relation to an SLA, as discussed in Section 2.4. Some behaviours will breach an SLA outright, indicating that one or more parties is no longer acting within the terms of the agreement. Other undesirable behaviours will be tolerable provided they are associated with some compensation. In the remainder of this chapter I refer to the former behaviours as *breaches* of the agreement, whereas the latter are *violations* of some condition in the agreement. A *condition* associates a behaviour of either the service or a party with a violation, which may be associated with a penalty, or with a breach of the agreement, so conditions may be either violated or breached. The agreement may be breached other than by a behaviour breaching a condition. Failure to conform to some behaviour implied by the agreement and represented by the domain model will also represent a breach of the agreement, for example an obligation to administer the SLA, as discussed below.

Any definition of reliability in an ASP SLA will rely on the notion of service failure, and I established in Chapter 2 the likely need for conditions relating to reliability. A well-established definition of failure within the dependable-computing community is described in [148], which I rely upon here: 'a *failure* is an event that occurs when the delivered service deviates from the correct service'. According to [148], a failure is due to an *error*, which is 'that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service'. Errors are hence features of the internal operation of a service, and are only relevant to SLAs when they affect the behaviour observable at a service interface. An error is presumed to occur due to a *fault* in a service, which may be permanent or transient, and which is 'the adjudged or hypothesised cause of an error'. Faults may be physical or informational, and may be caused by natural events, human errors or deliberate malicious behaviour. The consideration of faults may be relevant to assessing the risks involved in a particular service-provisioning scenario. However, failures, rather than faults, are relevant to the description of an SLA.

The definition of failure given by [148] relies on a notion of 'correct service'. The use of an SLA provides the participants in a service-provisioning relationship with the opportunity to define precisely what is meant by this: correct behaviour is any behaviour not identified as faulty in the SLA. However, not all failures imply violations or breaches of an SLA. Occasional failures may be acceptable to the client according to the conditions of the SLA.

In conclusion, failures, violations and breaches, are events that may be defined in relation to an SLA. However, there is no direct equivalence between these three types of event. The definition of failure used in this work is taken from authoritative prior work on dependable computing.

In any SLA, any violation or breach should eventually become apparent to some interested party. It will be necessary for a party making an assessment of a violation or a breach to do so on the basis of evidence that they have somehow obtained. In the case of a breach, it is not necessary for the SLA itself to define how this should occur – if one party breaches an SLA then the agreement has been broken overall, and the parties will be free to trade allegations on whatever basis they please. However, it is important to understand how violations will be calculated, as these do not terminate the agreement. To this end SLAng includes the syntactic and semantic elements shown in Figure 6.4.

An SLA contains a number of condition clauses. The class `ConditionClause` is abstract because a number of different types of conditions will be needed. It also extends the class `AuxiliaryClause`. Auxiliary clauses are statements in an SLA that may be optional and can be referred to from several other clauses. They are therefore notionally contained by the top-level SLA element, rather than a more specific type of clause.

The `Violation` class represents a record that a violation of a condition clause has occurred. It is a domain element, since such records are defined in relation to, rather than as part of, some SLA. However, a violation will refer to the clause in an SLA that has been violated, the identification of the party responsible for the violation, and the definition of the penalty that should be applied. The violation will be justified by a collection of evidence, which is the minimum such collection sufficient to establish

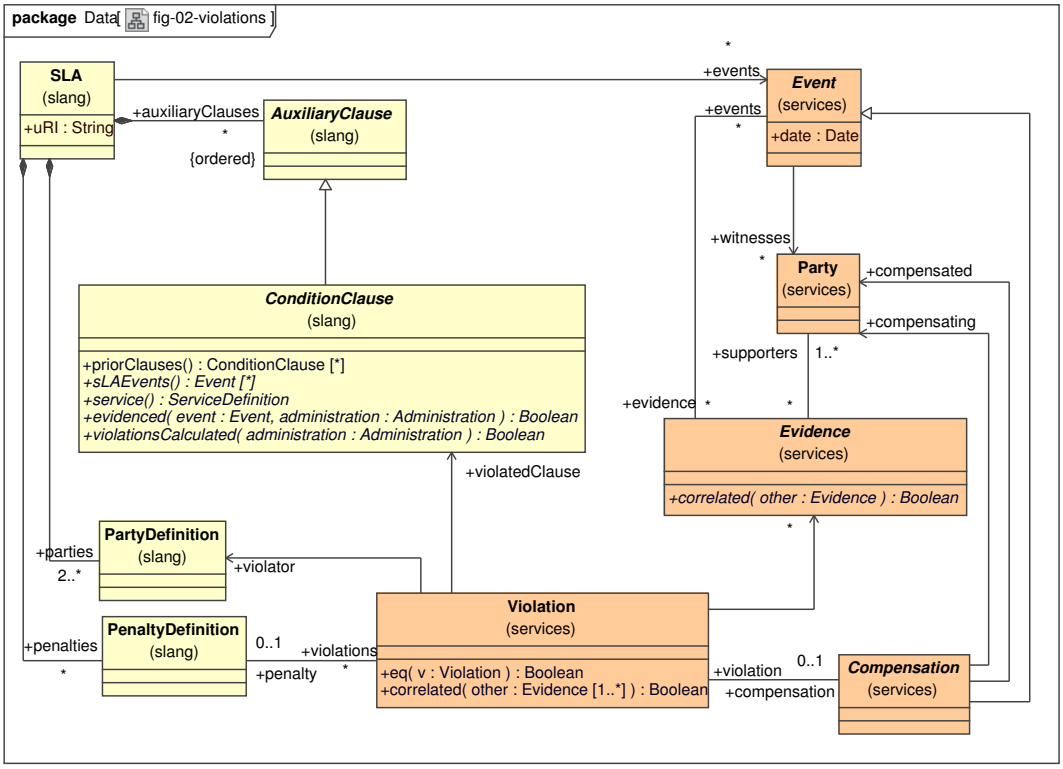


Figure 6.4: Condition definitions and the calculation of violations related to SLAng SLAs

that the violation has occurred. Evidence in turn is the record of some events. Events may be witnessed be some parties. Parties, perhaps as a result, but not necessarily, will lend their support to evidence. The evidence used to calculate violations of an SLA must have been gathered in relation to events that are pertinent to the SLA. There is hence an association between an SLA and the events that are relevant to it.

`Evidence` and `Event` are abstract classes in the model because the SLA will eventually have to define more specific types of events that are relevant, and the types of evidence with respect to which violations should be calculated. This is also related to the definition of specific types of condition clauses, since a particular type of condition clause will imply the relevance of certain behaviours to the SLA, and hence the evidence required to calculate violations of the clause. This is captured in the model by the inclusion of a number of abstract side-effect-free operations on the class `ConditionClause`: `slaEvents()` calculates what events are relevant to the clause; `services()` what services are relevant (useful for defining several obligations discussed below); `evidenced()` assesses whether adequate evidence has been collected in relation to a relevant event to make an assessment with respect to violations; and `violationsCalculated()` assesses, given a set of evidence and a collection of violations, whether violations have been correctly calculated in relation to the condition clause.

These abstract operations are used in invariants in the language specification to express various observations that should be true about all SLAs and condition clauses. These invariants provide an important component of the semantic definition of the language, namely the association between SLAs

and domain elements. For example, the events associated with an SLA should include all of the events relevant to the condition clauses that it contains. The formulation of this invariant is given in the next section.

This pattern of expressing invariants over abstract side-effect-free operations is used repeatedly in the SLAng language specification, and achieves two important things. First, it allows the language to capture knowledge that is true of all SLAs without having to define syntax to support all SLAs. Second, the abstract operations reveal explicitly what is known only in abstract to provide guidance for the extension of the language.

Associated with a violation may be the obligation for the violating party to perform some kind of compensating action, as represented by the abstract `Compensation` class. The specific details of these obligations may be encoded in extensions to SLAng using invariants defined in subclasses of `PenaltyDefinition` and `Compensation`. The `Compensation` class is also intended to support the definition of payment schemes.

6.5 Administration

In the previous section I presented semantic elements describing how violations should be calculated in relation to condition clauses. However, this is not sufficient to establish when the violations should be calculated, or by whom, an activity that I refer to as *administering* the SLA. As discussed in Section 2.5, it will frequently be necessary for an SLA to describe rules governing the administration of the SLA. Figure 6.5 depicts the support that is provided by SLAng for this using the abstract class `AdministrationClause`.

Administration clauses define rules for how the SLA should be administered. To establish how, the `Administration` class represents administrations in the domain model for the language. According to the model, administrations are events. One or more parties participate in an administration. Each party submits an account containing any evidence that they believe to be relevant to determining violations of the service. Somehow, based on this evidence, an agreed account of the behaviour of the service is produced, and on this basis violations are calculated. Each administration is associated with an administration clause (in an SLA) which provides a mandate for the administration and constraints over how it should be conducted.

Clearly there is great variety in when and how an SLA may be administered. Again, I capture this using a combination of abstract side-effect-free operations and invariants. The principle obligation expressed in the `SLA` class is that the parties must act in such a way as to ensure that all administration clauses included in the SLA are successfully administered. What this entails is delegated to the `AdministrationClause` class using the `administered()` operation, which assesses whether the collection of events associated with the SLA contains sufficient and correct administration events.

As discussed in Section 2.5, some SLAs may offer complete flexibility concerning administration, provided compensation is delivered within some time limit of violations occurring. Nevertheless, I have opted to make administration clauses a central element of SLAng SLAs in all cases. If flexibility is possible, it will improve the understandability and analysability of an SLA to make this explicit by

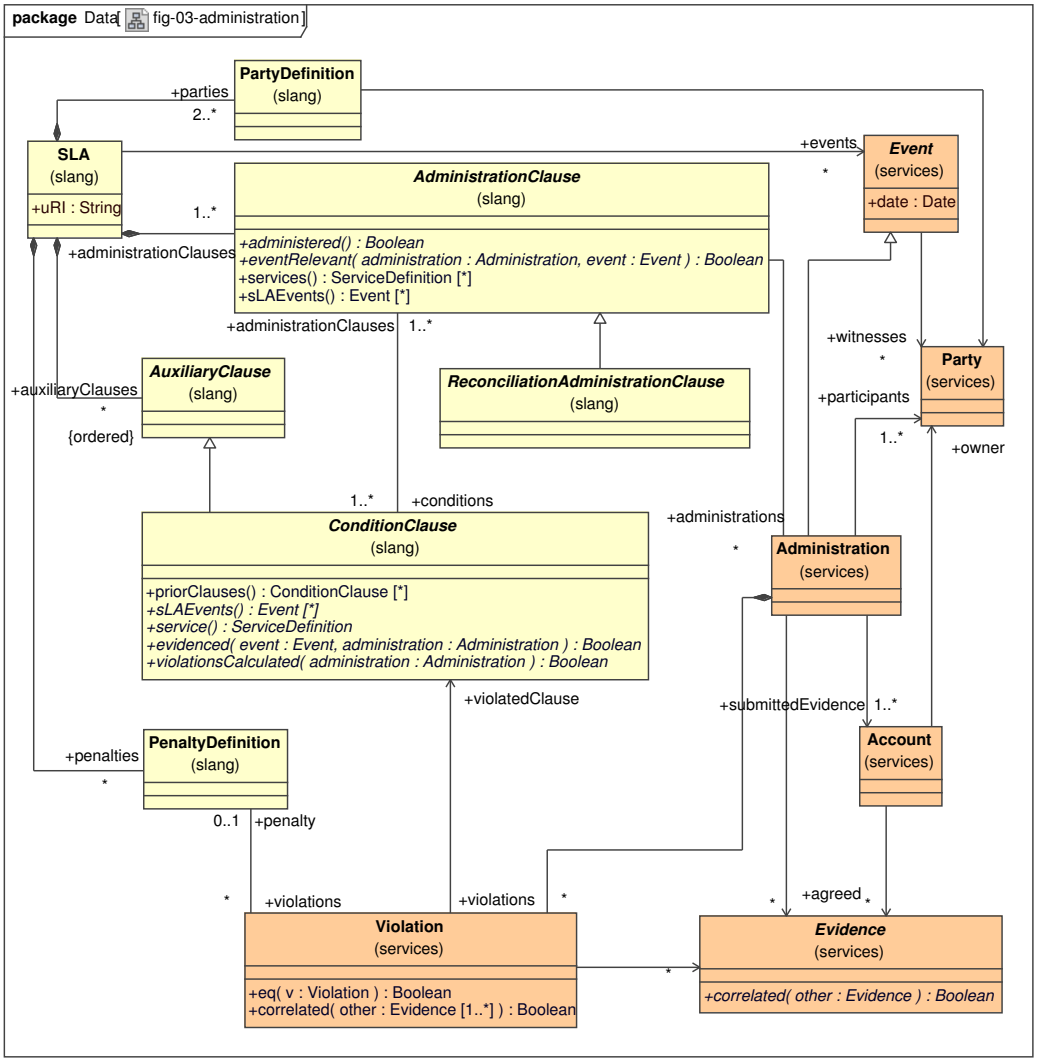


Figure 6.5: The administration of SLAng SLAs

encoding this in subclass of `AdministrationClause`.

The (abstract) meaning of SLAng SLAs is captured to a large extent using a set of invariants and side-effect-free operations defined over the `SLA` class and the `AdministrationClause` class, according to the model-denotational approach. An SLA is associated with a set of pertinent domain events, which are the events associated with administering the SLA. This is expressed using the following invariant over the `SLA` class:

```
events = administrationClauses.sLAEvents()->asSet()
```

The operation `sLAEvents()` on class `AdministrationClause` is defined as follows:

```
conditions.sLAEvents()->union(administrations)->asSet()
```

In other words, the relevant events are any events relevant to a condition being administered, plus the events representing the administrations themselves. It is then necessary to state that this continuum of events must conform to the intent of the SLA. This is stated by asserting that the SLA must be

administered according to the standards of the administration clauses that it includes (an invariant on class SLA):

```
administrationClauses->forall(a : AdministrationClause |

  a.administered()
)
```

The operation administered() on class AdministrationClause is abstract, because the behaviour of parties in administering SLAs may be acceptable according to various standards. However, given that this invariant must hold, the following invariant on AdministrationClause requires that violations of the SLA be calculated according to the standards of the conditions associated with an administration clause.

```
administrations->forall(a : ::services::Administration |

  conditions->forall(violationsCalculated(a))
)
```

The meaning of specific types of conditions, for example reliability, throughput and timeliness, is therefore largely established by overriding the violationsCalculated() operation in subclasses of ConditionClause.

Finally, administration clauses define an important obligations for the participants in an SLA. The monitoring obligation states that if an event is relevant to a condition clause being administered, then sufficient evidence related to that event must be presented at the administration to determine violations of the relevant conditions (relying on the sLAEvents() and evidenced() operations on ConditionClause):

```
let
events = sLA.events
in
administrations->forall(a : ::services::Administration |

  events->forall(e : ::services::Event |

    sLAEvents()->includes(e)
    and
    eventRelevant(a, e)
    implies
    conditions->forall(c : ConditionClause |

      c.sLAEvents()->includes(e)
      implies
      c.evidenced(e, a)
    )
  )
)
```

Invariants in the domain model, and accuracy constraints (discussed below), require this evidence to faithfully represent the events that are pertinent to the SLA.

A `ReconciliationAdministrationClause` is potentially useful in mutually-monitorable SLAs. These clauses require administration in which both the client and provider of any services associated with condition clauses being administered participate and submit evidence. In Chapter 5 I anticipated that such reconciliations may be needed in the event of a disagreement between the parties as to the behaviour of the service.

6.6 Accuracy of evidence

As discussed in Section 5.2, in the case of mutually monitorable SLAs, it is desirable to establish rules concerning the accuracy of the measurements forming the basis for the assessment of a violation. This is also true of less monitorable types of SLAs, although it may not be possible to check whether any accuracy constraint is being adhered to in that case. In Chapter 5 I describe an approximately monitorable accuracy constraint appropriate to this purpose. Figure 6.6 depicts the inclusion of this constraint in the SLAng language specification.

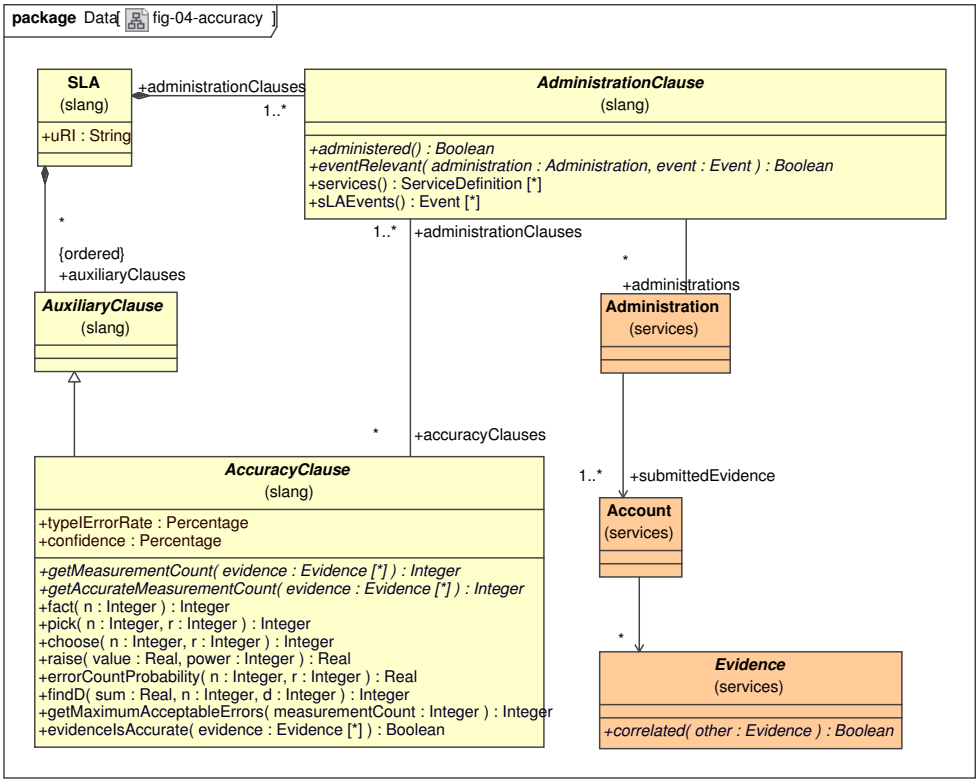


Figure 6.6: Accuracy constraints in the SLAng langauge

The constraint is represented by the abstract class `AccuracyClause`. All accuracy clauses according to my design of the accuracy constraint specify a type I error rate for the constraint, which is the proportion of times that a participant submitting an honest log will be accused of cheating, and a confidence measure, which is the confidence stated by the participants that any given measurement submitted will meet the standard of accuracy to which it is held. Given these values, all that is needed to assess whether the constraint has been met are values for n, the total number of measurements in the log under consideration, and d, the total number of incorrect measurements. At this level of abstraction the

language does not make any assumptions about the type of measurements being considered, or how their accuracy may be assessed. The provision of values for n and d are therefore delegated to more specific types of accuracy clause using the abstract operations `getMeasurementCount()` (returning n) and `getAccurateMeasurementCount()` (which returns $n - d$).

Note that this constraint can never be evaluated with certainty, since the true value returned by `getAccurateMeasurementCount()` cannot be known with certainty. However, since the domain model contains classes for both evidence and events, and evidence is notionally associated with the events that it documents, it is possible to precisely define, for a particular type of evidence, what is meant by an accurate measurement. The operation `evidenceIsAccurate()` operation then represents the effect of the constraint over an arbitrary log of evidence. Accuracy clauses are associated with administration clauses and constrain the evidence presented in support of an administration to be accurate. This is captured by an invariant on the class `AdministrationClause`. Given a trusted log of evidence correlated to that submitted during administration, with known error characteristics, conformance to the accuracy constraint can be approximately monitored using the statistical hypothesis described in Section 5.2.2.

The use of a constraint on the accuracy of evidence used to assess violations conveniently avoids the need to consider how the residual error in the measurements propagates through the calculation of violations, the need for which is discussed in Section 2.7.4. Providing the evidence used is sufficiently accurate, the parties agree to having the violations calculated according to the semantics of SLAng.

6.7 Termination of SLAs

Two fundamental requirements for systems of SLAs, listed in Section 2.7 indicate the need to include conditions related to termination of the SLA. Requirement SLA 3 states that a party should become entitled to receive compensation when an SLA is terminated by a peer. Requirement SLA 4 states that a party should have the right to terminate an SLA without penalty if their peer's behaviour becomes unacceptable.

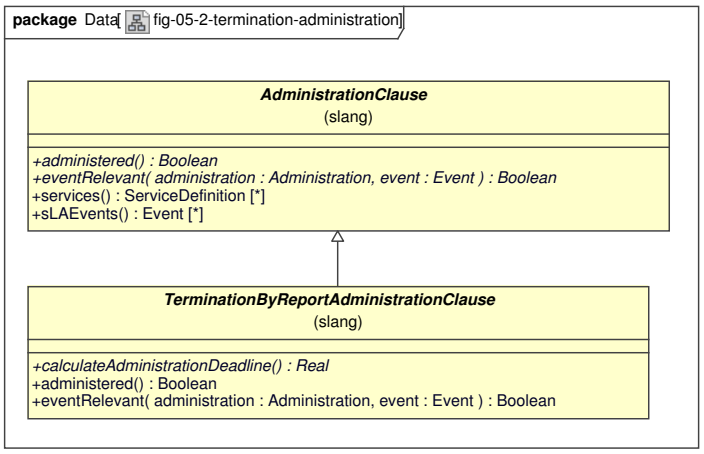


Figure 6.7: Clauses governing the final administration of a terminated SLAng SLA

Figure 6.8 shows the support SLAng offers for describing the conditions under which an SLA

terminates. Terminating condition clauses terminate the SLA if they are ever violated. This must be respected when checking whether administration clauses have been administered, and support for determining whether a terminating condition has previously been violated is built into the `AdministrationClause` class.

Parties should entering into an SLA should anticipate the possible need to terminate the SLA. If they wish to limit the possible consequences of doing so, then an SLA should include prearranged provisions relating to termination. I assume that if a party is going to unilaterally terminate an SLA in a prearranged manner (rather than by merely commencing to ignore the provisions of the SLA) then they will have to at least notify their peer. Termination under these circumstances is covered by instances of the `TerminationByReportConditionClause`. Defining the semantics of statements of this type relies on some extra exposition in the domain model. Reports are types of events that represent communication between the parties. A termination report is a communication that indicates the intent to terminate an SLA on the part of the dispatcher. A termination-by-report condition clause is violated based on evidence of a termination report applying to the SLA in which the clause is written.

`TerminationByReportConditionClause` is an abstract class because it is necessary to specify the scheme by which penalties should be calculated in these circumstances. However, it makes concrete the operations `sLAEvents()` – termination reports referencing the SLA are relevant – `service()` – no specific services are relevant – `evidenced()` – termination reports are evidenced by having an associated report record – and `violationsCalculated()` – a violation should be calculated citing the originator of the termination report as the violator, the report record as the evidence, and the penalty calculated by `calculatePenalty()` as the penalty. This last rule is expressed using the following OCL definition of `violationsCalculated()`:

```
let
agreed = administration.agreed,
violations = administration.violations
in
agreed->select(
  oclIsKindOf(::services::ReportRecord)
  and
  oclAsType(::services::ReportRecord).report.oclIsKindOf(
    ::services::TerminationReport)
)->forall(e : ::services::Evidence |

    violations->exists(v : ::services::Violation |
      v.violator =
        sLA.parties->any(
          party = oclAsType(::services::ReportRecord
                    ).report.dispatcher)
    and
    v.violatedClause = self
    and
    v.evidence = Set(::services::Evidence) { e }
    and
    v.penalty = calculatePenalty(
      e.oclAsType(::services::ReportRecord), agreed)
  )
)
```

This provides a succinct example as to how the effect of conditions can be expressed by extensions to the `ConditionClause` class, and if necessary, extensions to the domain model.

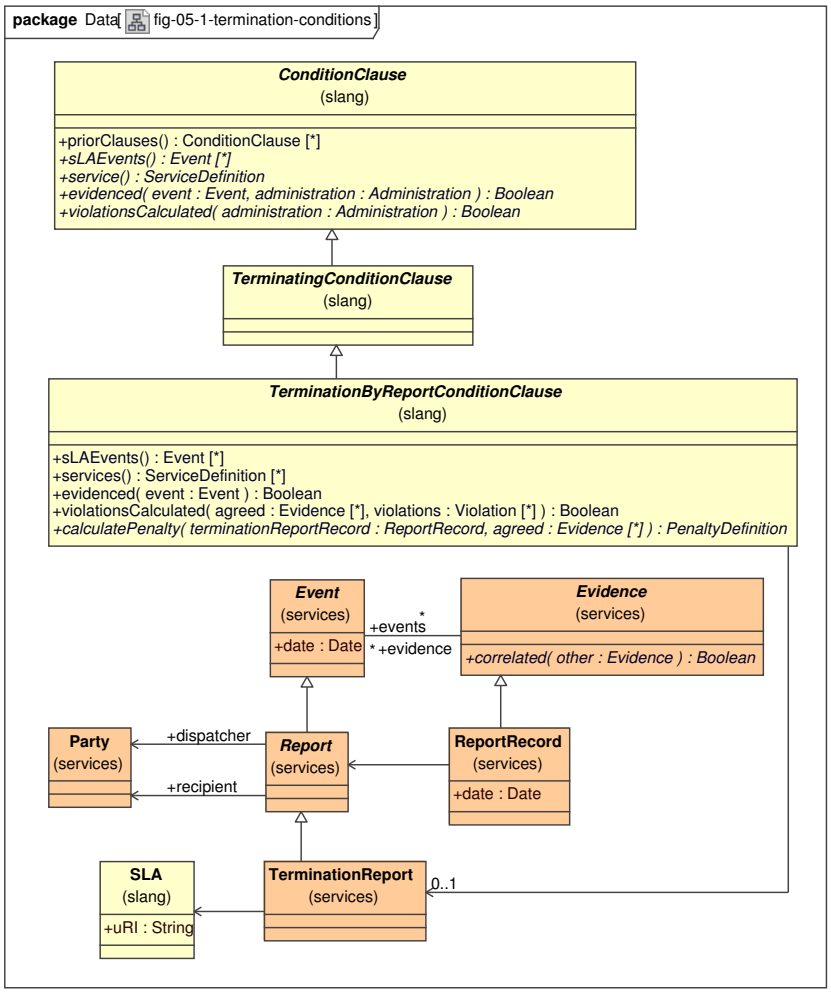


Figure 6.8: Conditions and semantics related to the termination of SLAng SLAs

If a party issues a termination report then they are unlikely to wish to wait for the next routine administration for the SLA to occur. Assuming a termination report will not otherwise trigger an immediate administration, the parties will probably wish to state that it does. Support for this is implemented by the class `TerminationByReportAdministrationClause` which requires that a final administration of the SLA takes place within some deadline of a termination report being exchanged. The class, shown in Figure 6.7, is abstract, since the length of this deadline may need to be calculated based on other factors pertaining to the SLA.

`TerminationByReportConditionClause` extends `TerminatingConditionClause` to indicate that any violation of this clause signals the imminent end of the agreement. Ordinary administration clauses must not require administrations to occur after such a violation has been calculated.

The time at which a termination report is exchanged will be a matter of concern to the parties to the SLA being terminated, as it will determine to what point they must adhere to the usual conditions of the SLA. As usual, this will be captured by evidence, which may be somewhat inaccurate. The parties will

wish to constrain this accuracy, which may be achieved by a `ReportRecordingAccuracyClause`, shown in Figure 6.9. Such a clause governs the accuracy of exchange of any type of report. This generalisation has been included because other types of condition may be related to an exchange of reports, as described below in relation to availability conditions. This class is abstract, because the accuracy required for a particular report may depend on factors that are hard to anticipate, such as the state of the service. However, it is likely that a constant basic standard of accuracy will be desired, so I have also included in the core language the class `PermanentFixedReportRecordingAccuracyClause` that can be used to establish this.

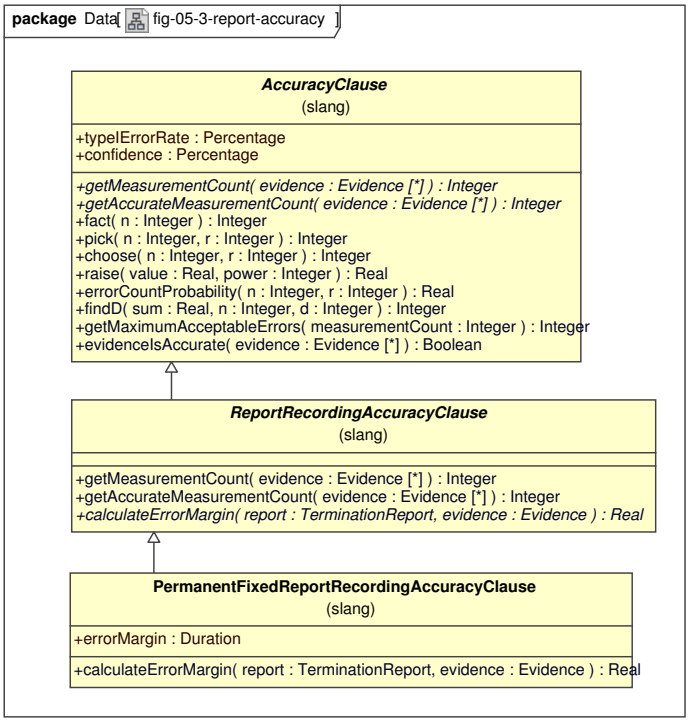


Figure 6.9: Accuracy clauses governing the recording of the exchange of reports related to SLAng SLAs

6.8 Electronic services

I now move to the description of syntactic and semantic provisions in the language that are related to electronic services. In order to express conditions related to an electronic service in an SLA it will be first necessary to describe it. This is achieved in SLAng using the `ElectronicServiceDefinition` class, a concrete refinement of the more general `ServiceDefinition` class. The relevant classes are shown in Figure 6.10.

An electronic service consists of the provision of access to one or more electronic-service interfaces to one or more electronic-service clients, which are defined to be software capable of accessing the interface. All of the interfaces must be owned by one party, the provider, and likewise all of the instances of client software must be owned by another, the client. Electronic-service interfaces consist of operations with known parameter types. Parameters may be associated with requests (`IN` parameters), responses (`OUT` parameters) or both (`IN_OUT` parameters).

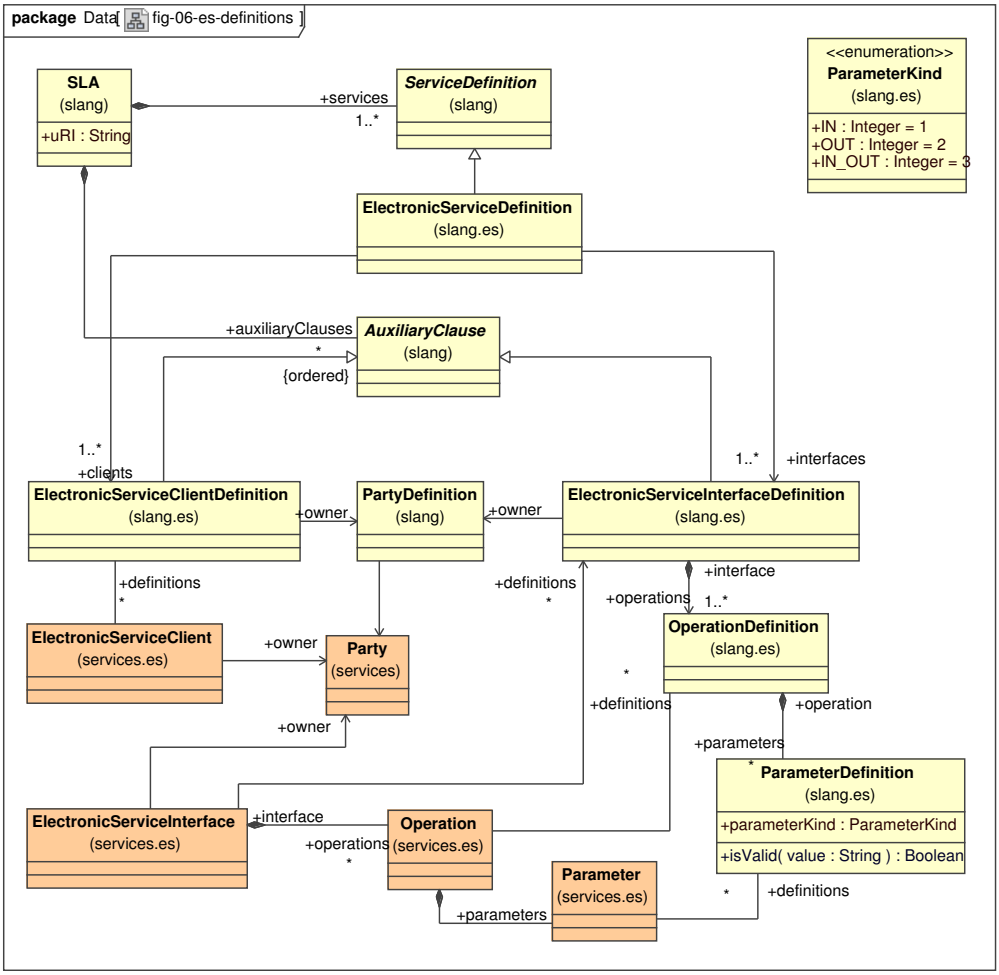


Figure 6.10: Definitions of electronic services in SLAng, and corresponding semantic elements

For an SLA to be valid, an electronic service conforming to the elements of the electronic-service description must exist in the real world. This is represented by the domain elements corresponding to the syntactic definition of the service.

Requirement SLA 1 includes the provision that a system of SLAs for ASP should be able express conditions over the timeliness and reliability of an electronic service. To describe the semantics of such conditions, it will be necessary to have a reference model of the behaviour of electronic services, and the way that evidence pertaining to that behaviour should be collected. These elements are shown in Figure 6.11. Electronic-service clients issue service requests to operations of an interface. These may include a number of parameter values. Depending on the functioning of the service, and whether a fault occurs, these may result in service responses, which may also carry parameter values. The monitoring of such an episode is presumed to result in a service-usage record, which records the moment that the request was issued, the interval between the request being issued and the response being fully received, what operation was invoked by what client, and what parameter values were exchanged.

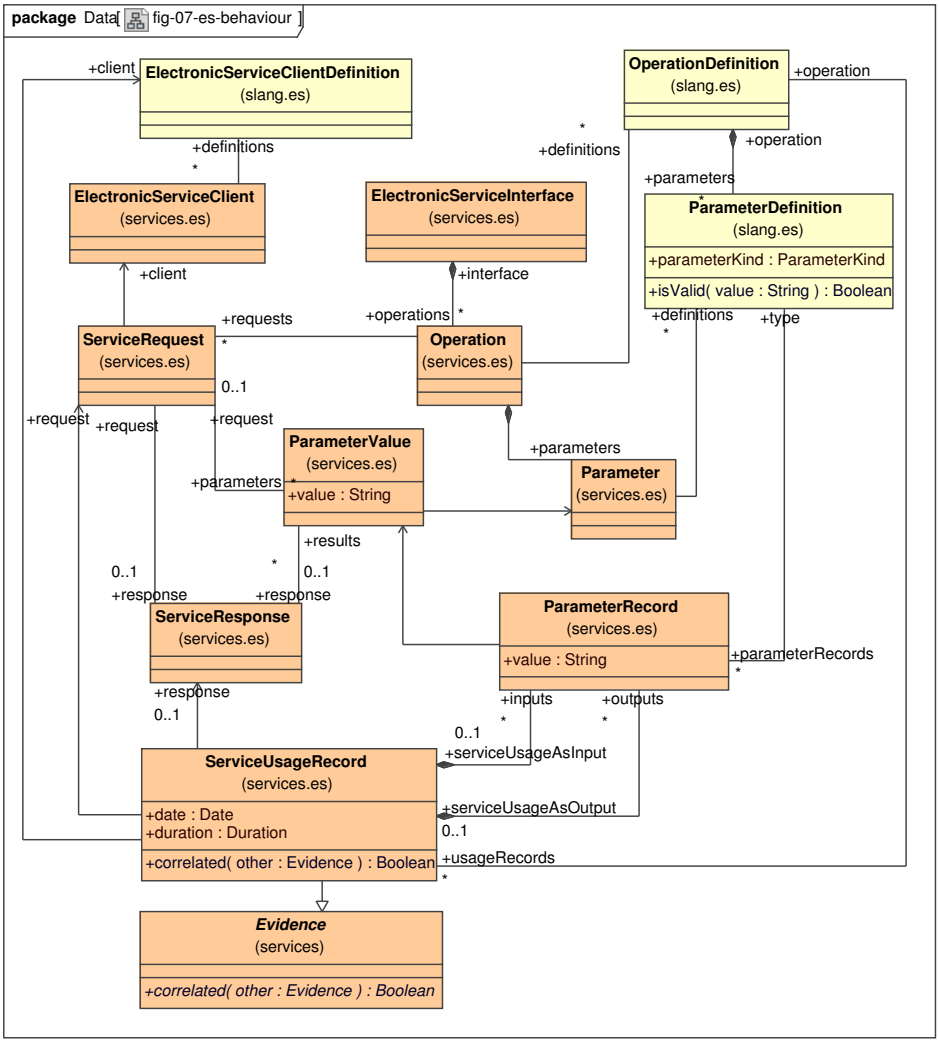


Figure 6.11: The behaviour of electronic services, assumed by the SLAng semantics

6.9 Reliability, timeliness and throughput conditions

6.9.1 Service behaviour restrictions

In Chapter 2, three particular kinds of condition are identified as being necessary in ASP SLAs. These are reliability, latency and throughput. SLAng provides support for all three kinds of condition through the use of a single, more general type of condition, the service behaviour restriction.

Consider what is characteristic of a reliability condition over an electronic service: [148] defines reliability as ‘continuity of correct service’. Adopting this definition requires a definition of correct service, or equivalently, incorrect service, which as stated above may be established in an SLA. Correct or incorrect service behaviour will have to be defined with reference to a functional definition of the service. However, individual failures may not be problematic to the client, so this is not enough to define reliability alone. Multiple failures might be problematic however, if they occur sufficiently closely together to be disruptive to the client, so a reasonable approach to defining a reliability constraint is to restrict the maximum number of failures that may be seen within a sliding window of time of a specified length. Such a definition would also accommodate the case that no failures were tolerable, as

the maximum number could be set to zero.

Now consider how a latency constraint may be defined: operations that take too long to complete are bad, but how bad are they? If they take far too long, the client will have to assume that they will never complete at all, and act accordingly, so this kind of latency violation will be equivalent to a failure. Occasional slow operations may be tolerable, but persistent slowness will be problematic. This description of latency constraints is very like the description of reliability given above, so again, restricting the number of slow operations in a sliding window seems appropriate.

Finally, consider throughput restrictions. Most middleware systems used to implement electronic services accommodate a degree of concurrency in the submission of requests. This is particularly important when several distributed electronic service client programs will access the service concurrently, as it is undesirable for the client party to have to guarantee that no two requests will arrive too close together. Therefore some amount of high throughput is acceptable. Throughput becomes unacceptable when a large number of requests arrive in a short interval, overloading input queues, or causing bottlenecks at contended resources, thereby causing failures or resulting in high latency for service requests, potentially making the service provider liable to pay penalties. Hence, once again, it is appropriate to constrain throughput as a maximum number of requests that may be submitted in a specific interval.

Since all three cases are so similar, it is possible to generalise. Essentially failures, delays and requests represent possible behaviours that may be manifest in the service and the behaviour of the parties with respect to the service. These behaviours may be tolerable in moderation, but if too many happen in too short a time, they become intolerable.

It is also possible to see the usefulness of constraints of this type when applied to service behaviours not uniquely associated with electronic-service behaviours. A groceries-delivery service may occasionally deliver vegetables that are not in prime condition. Their client, a restaurant, can cope with this by varying their menu from time to time. However, if it occurs too frequently the reputation of the restaurant for delivering fresh food may suffer, so the restaurant may wish to penalise this in an SLA.

Any kind of activity as part of a service may be assessed based on its measurable qualities, and classified as either acceptable or undesirable. Being undesirable need not be intolerable however, so a constraint may instead be needed that undesirable outcomes do not occur too frequently. Since this pattern is so generally applicable, I have implemented support for conditions of this kind as part of the generic syntax package of SLAng.

Figure 6.12 shows the classes `ServiceBehaviourRestrictionConditionClause` and `ServiceBehaviourDefinition` supporting the specification of clauses of this kind. A service behaviour definition will define a particular type of behaviour, and state what party is responsible for it occurring. Given a collection of evidence, it will be capable of identifying instances of the behaviour indicated by the evidence, and when these instances are deemed to have taken place. A service behaviour restriction clause associates penalties with too many instances of the behaviour occurring in too short a time period. `ServiceBehaviourRestrictionConditionClause` is abstract because it is necessary to determine the desired width of time window at and number of occurrences of the behaviour

acceptable at any given point in time.

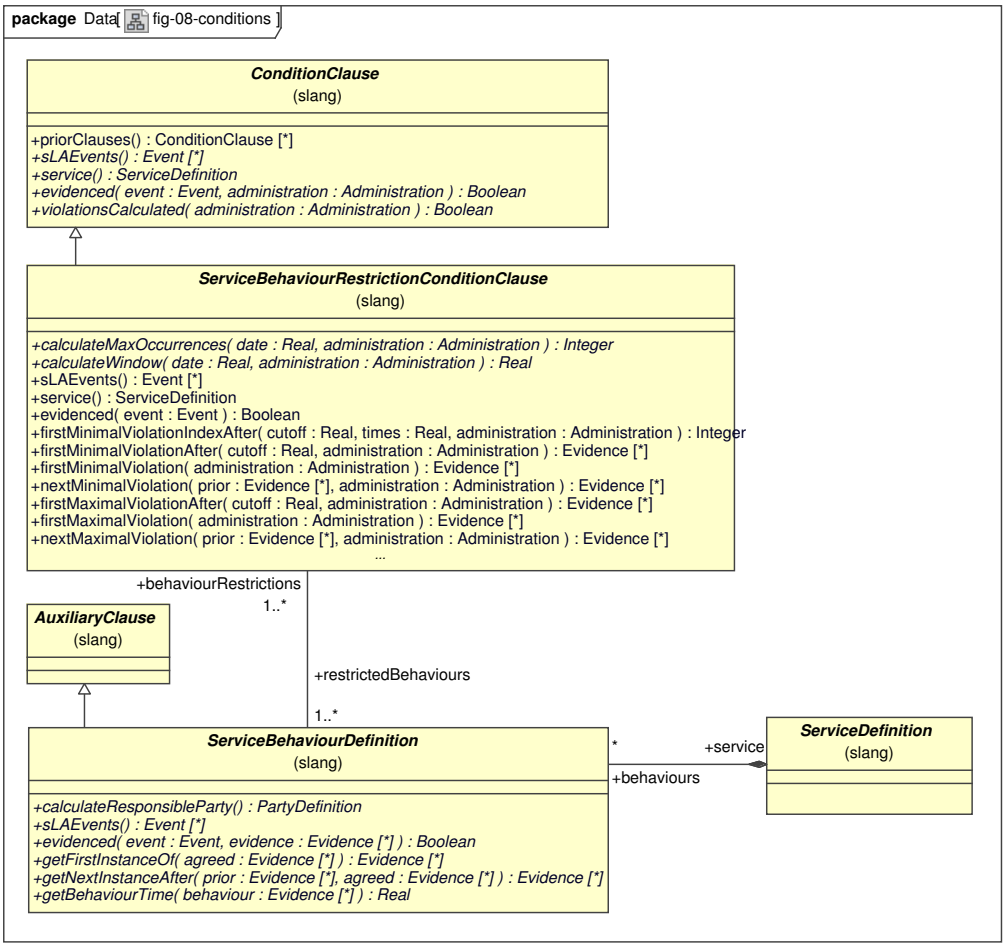


Figure 6.12: Clauses supporting conditions related to restrictions on service behaviours

Once these parameters have been determined, penalties can be calculated based on two types of analysis of the evidence constituting a violation. A minimal violation is the least amount of evidence required to establish that a service behaviour restriction has been violated, and given a set of evidence, the `ServiceBehaviourRestrictionConditionClause` class provides operations to sequentially examine each minimal violation. On the other hand, two or more minimal violations that occur so closely together that any superimposition of the sliding window between the time of the first violation and that of the last will contain a set of evidence representing a minimal violation may be regarded as one continuous incident of violation of the clause. The `ServiceBehaviourRestrictionConditionClause` therefore also provides operations capable of iterating over all of the maximal violations contained in a set of evidence, where a maximal violation is any consecutive period where any superimposition of the time window will contain a minimal violation, and no other period with the same properties both overlaps and is longer.

Building on this foundation, more specific support for reliability, latency and throughput conditions can be provided. This is discussed in the following section.

6.9.2 Electronic-service usage behaviour definitions

Two requirements for SLAs are that they are precise in their meaning, and that they allow the client to receive compensation in the event of the service proving unreliable. This implies that SLAs should precisely define what is meant by reliability, and because the notion of reliability depends on the notion of failure, what is meant by failure must also be precisely defined. This suggests that services in relation to which precise SLAs are made will require formal definitions.

It is clear that extensions to the class `ServiceBehaviourDefinition` can be used to provide a formal definition of the functional behaviour of a service, as well as properties related to the timing of requests and responses. Service behaviour definitions may be related to the domain model of service behaviour either directly or via a relationship with a service definition clause. Since the domain model for electronic services captures details of service usages and their parameters, it is possible to encode relationships between input and output parameters capturing the functional behaviour of the service in side-effect-free operations specified on a subclass of `ServiceBehaviourDefinition`. The extended clause is therefore able to identify instances in which this behaviour was violated.

Since reliability conditions will be commonly required, relying on this facility implies that extensions to SLAng to formalise service behaviour will usually be required when writing an SLA for a new type of service. This is perfectly consistent with the approach followed in the design of SLAng, which aims to deliver an abstract, extensible language. However, there is a qualitative difference between the extensions required by the language so far, and extensions required to define the functionality of the service. This is that all extension points defined thus far, for example, to calculate penalties or or determine when an SLA should be administered, have the potential to be extended in a service-independent manner.

Service-independent vocabulary clearly has a higher potential for reuse across SLAs than service-dependent vocabulary. This suggests that over time, a richer vocabulary of concrete syntax elements might be added to the core language in order to increase its expected adequacy. If any given extension point in the language had the potential to be extended to provide concrete syntax in a service-independent manner, it admits the possibility that a core language expanded with generic concrete syntax might be completely adequate without further extension to some new SLA. If I follow the approach of insisting that any reliability constraint requires extension to the language in a service-specific manner, this potential is greatly reduced for ASP SLAs.

Another major problem with insisting that ASP services have a formal functional specification is that in the domain of application services, few services are formally specified as a matter of course. An absolute requirement for a formal specification of every service using an SLA may increase the expense of entering into an SLA to the point that they are no longer an attractive method for balancing risk.

Finally, even if a formal specification of a service exists prior to the negotiation of an SLA, it may not be encoded in OCL. Re-modelling the functional behaviour in a language extension may be an unnecessary expense.

These considerations combine to suggest the need in some SLAs for a certain amount

of informality in the definition of failures of a service. I provide several classes supporting this. The first is `ElectronicServiceUsageBehaviourDefinition` which is a subtype of `ServiceBehaviourDefinition`. This class may be extended to provide vocabulary to define behaviours observable in individual service usages. Crucially, the semantic element `ServiceUsageRecord` is augmented with the property `behaviours`, implying that when reporting on service usages during administration, parties must explicitly identify, for each service usage, the predefined service-usage behaviour definitions to which the service usage conforms. These classes are shown in Figure 6.13.

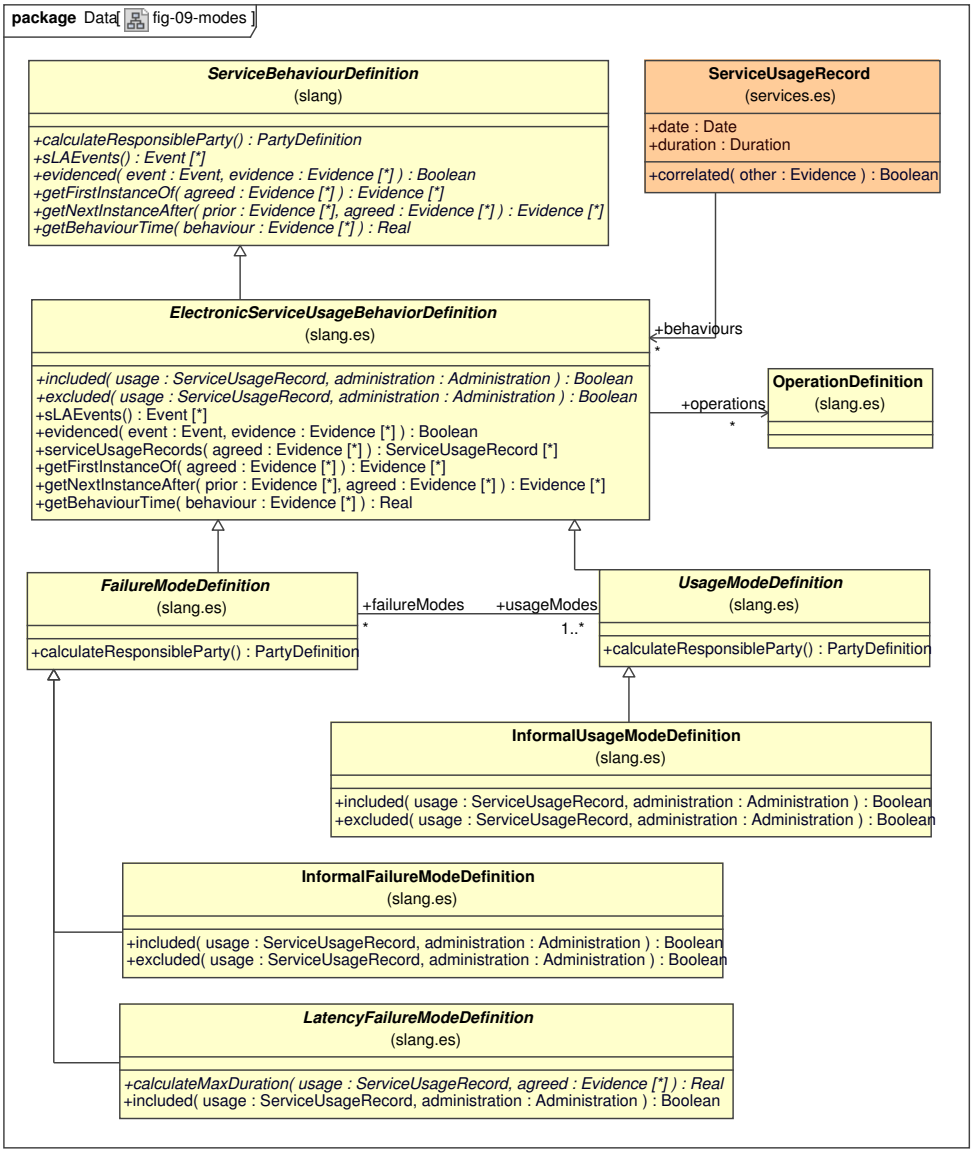


Figure 6.13: Service behaviours relevant to reliability, timeliness and availability conditions

In reliability conditions, the service behaviour being restricted is that which is exhibited by failures. In latency conditions over synchronous operations these failures are characterised by a delay in the production of the result of operation. In throughput conditions, client behaviour that includes a request

to the service is being restricted. When specifying a throughput condition, it is necessary to consider what usages will be relevant to the clause. Often this may be straightforwardly defined as any operation invocation on some service interface. However, occasionally it may be more complicated. The resource utilisation of an operation may be heavily dependent on the parameter values passed to it, so it may be desirable to restrict usage of the service on this basis. Similarly to failures, there may be requirements to specify types of usages both formally and informally depending on the circumstances.

These types of behaviour are all most commonly exhibited by individual service requests and hence can be captured using extensions to `ElectronicServiceUsageBehaviourDefinition` (if failures or patterns of usages emerging over multiple usages must be restricted then `ServiceBehaviourDefinition` can be extended instead). However, further generic refinements of this class are possible to capture in common knowledge concerning these types of behaviour in the core definition of SLAng.

As behaviours, the principal distinction between usages and failures, is that the party responsible for causing them is the client (the party controlling the client software) in the case of the usages, and the provider (the party providing the electronic service) in the case of failures. This distinction is captured in the provision of two subclasses of `ElectronicServiceUsageBehaviourDefinition`, `UsageModeDefinition` and `FailureModeDefinition`. Relationships between these modes can also be described, to establish the possibility of failures in a particular failure-mode occurring as a result of a request in a particular usage-mode. This is important in relation to availability conditions, as described below.

Further to this, concrete extensions to `ElectronicServiceUsageBehaviourDefinition` must override two side-effect-free operations, `included()` and `excluded()`. An invariant on `ElectronicServiceUsageBehaviourDefinition` asserts that a definition of that type must be referenced by a service usage if these operations deem that it should be included, and not excluded.

This allows the definition of an abstract latency failure-mode, which asserts that a service usage should be reference definitions of that type whenever the usage takes longer than some duration (obtained from an abstract side-effect-free operation). However, there may also be times when a usage should be excluded despite this condition being met. This may be specified by inheriting the clause type and overriding the `excluded()` operation.

As discussed above, the decision to consider a service-usage as being an example of a particular electronic-service behaviour may be subjective, or rely upon standards expressed externally to the SLA. Therefore, I have included the syntactic types `InformalUsageModeDefinition` and `InformalFailureModeDefinition` in the language. These classes define both `included()` and `excluded()`. In both cases, a service-usage should reference an instance of a clause of either type only if it does, and not otherwise. This circular definition leaves the referencing of these clauses to the discretion of a party participating in an administration of the SLA. However, definitive natural language descriptions associated with the definitions of these clause types oblige the parties to consider any natural-language description of a behaviour included in the SLA when determining whether a service-

usage conforms to such a behaviour definition. Such clauses should of course be used with care, as the precision of the SLA will depend on the precision with which failures are described.

At this point it is appropriate to observe that all three types of conditions described here are mutually monitorable, because assessing violations of these conditions only relies on evidence being gathered relating to service requests and responses, both of which are events visible to both the client and the provider of the service.

6.9.3 Service-usage record accuracy

When implemented using service-behaviour restrictions, each of the three required types of condition, reliability, timeliness and throughput, rely on determining whether service usages with particular characteristics occur within a particular time window. Clearly this cannot be assessed with any kind of confidence unless some standard for the accuracy of measuring the moment of occurrence of a usage is established in the SLA. This is achieved by another type of accuracy clause, for service usage records, as shown in Figure 6.14. `ServiceUsageRecordAccuracyClause` supports the specification of error margins for date and duration measurements. `PermanentFixedServiceUsageRecordAccuracyClause` is provided to allow the specification of a permanent basic standard of accuracy for these measurements. If an `ElectronicServiceUsageBehaviourDefinition` is associated with a service behaviour restriction referred to by an administration clause, then an invariant ensures that a `PermanentFixedServiceUsageRecordAccuracyClause` is also specified in order to ensure that some basic standard of accuracy is agreed. This is necessary as service usage record accuracy clauses for specialised purposes may not assess the accuracy of all service-usages. This is an example of an invariant used entirely within the syntactic types, therefore providing restrictiveness in the language to prevent the specification of bad SLAs.

Service-usage clauses are approximately mutually-monitorable.

6.10 Availability conditions

Availability of a service is defined in [148] as ‘readiness for correct service’. It is common for SLAs to include constraints on availability, even though as I have observed, the availability of the service is not a prime concern of the client’s (who should only care if the service is available when they try to use it, hence resulting in a definition for reliability), neither is it monitorable by the client. Hence I have avoided providing support in SLAng for conditions related to availability as it is conventionally defined.

Unfortunately, there are potential problems with relying on reliability conditions alone to mitigate risks related to the functional quality of the service. These problems are related to the exploitability of the SLA.

It is possible that over the course of several interactions with an electronic service, the client will discover a combination of operation parameters that unless reparative action is taken by the service provider will always cause the service to fail. This is undesirable from the provider’s point of view, as the client (if they are not in a particular rush to process the particular parameters) may choose to reinvoke the operation on any occasion that it has some free input throughput, thereby increasing their likelihood of receiving compensation for unreliability from the service provider. However, if the client

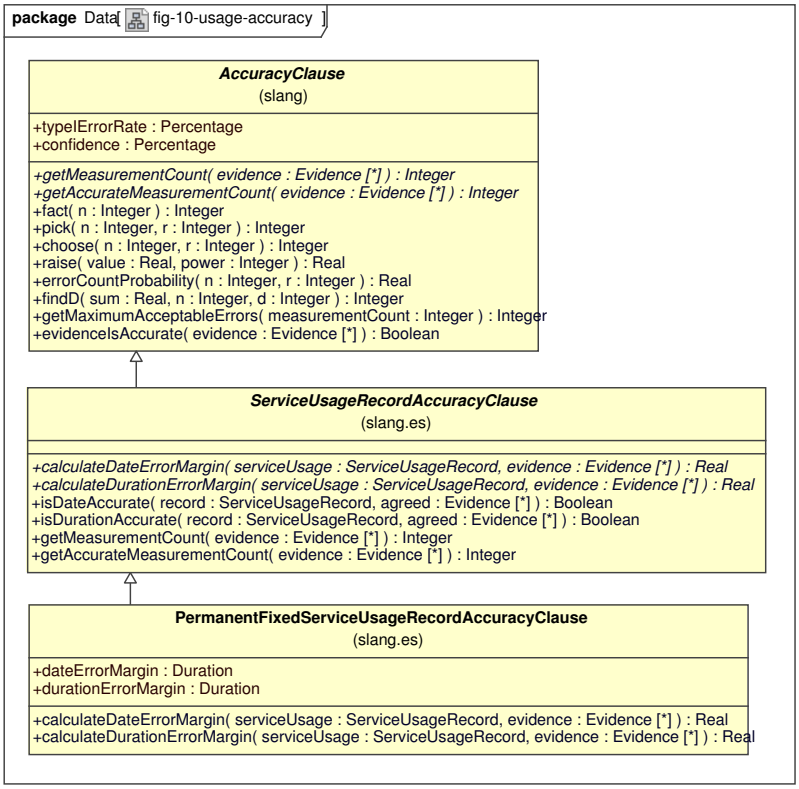


Figure 6.14: Clauses constraining the accuracy of reporting of service usages

needs to process the parameters urgently, then it is also undesirable from the client's point of view. This behaviour may become apparent to the service immediately, or it may not be noticed until reported at a later administration of the SLA.

The solution implemented in SLAng is to assume that the service provider and the client (as identified by the SLA, rather than the original provider and ultimate client in the scenario) have at least one channel of communication available between them besides the use of the service. I have already made this assumption in the modelling of termination reports in Section 6.7. Over this channel the parties can communicate bug reports and the service provider can issue bug fix reports. The period between a bug report and a bug-fix report being submitted can be regarded as a a period during which the service is unavailable in some category of usage in which the bug manifests itself. The relationship between failure modes and the usage mode in which they are presumed to occur is established in the SLA (see Figure 6.13).

Classes supporting the definition of availability conditions are shown in Figure 6.15.

Availability clauses in SLAng are a type of condition clause which are distinct from service behaviour restrictions. They are associated with a single usage-mode description, and a set of service-behaviour restriction conditions, which must implement reliability conditions (i.e. be associated with failure-mode definitions).

Either party may submit a bug report identifying a usage-mode associated with an availability condition clause. If the service provider submits the report then a period of unavailability begins automatically.

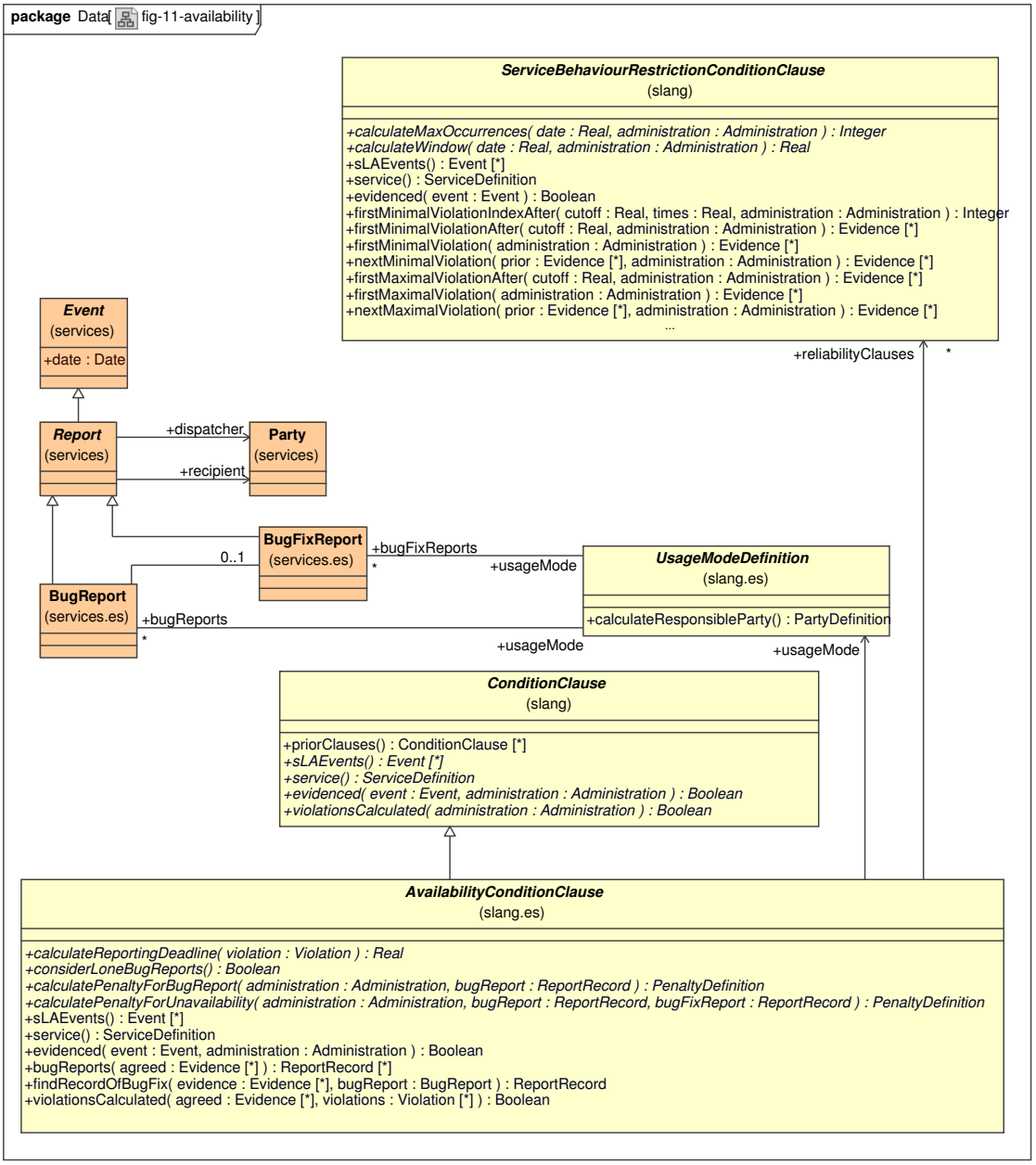


Figure 6.15: Availability clauses and supporting semantics

The client may submit the report within a time limit specified in the availability clause following a period of unreliability, as defined by one of the reliability conditions associated with the availability clause. Clearly, the reliability conditions must only be associated with failure-modes, the failures of which may occur in the usage-mode associated with the availability clause, and this is enforced using an invariant on the `AvailabilityConditionClause` class.

Availability conditions in a SLAng SLA can be used to protect the service provider from repeated requests by the client. The service provider may issue a bug report at any time, preventing the client from accumulating compensation related to unreliability, although at the cost of incurring a penalty for unavailability. The client is also afforded a degree of protection over a service provider who does

not mend their service when it malfunctions, since they have the opportunity to report problems and receive compensation if they are not fixed in a timely manner, making them equivalent to time-to-repair constraints in other languages.

The timing of exchanges of bug and bug-fix reports is relevant in determining penalties. Therefore the accuracy of report records for this purpose must be constrained for any SLA administration containing availability conditions, as described in Section 6.7.

The addition of the concepts of bug and bug-fix reports to SLAng is typical of the kind of flexibility that is occasionally required in expressing SLAs, and supportive of the notion that a language for SLAs must be highly expressive or extensible.

Since the client and the provider must exchange bug and big-fix reports to establish periods of unavailability, availability constraints are mutually monitorable.

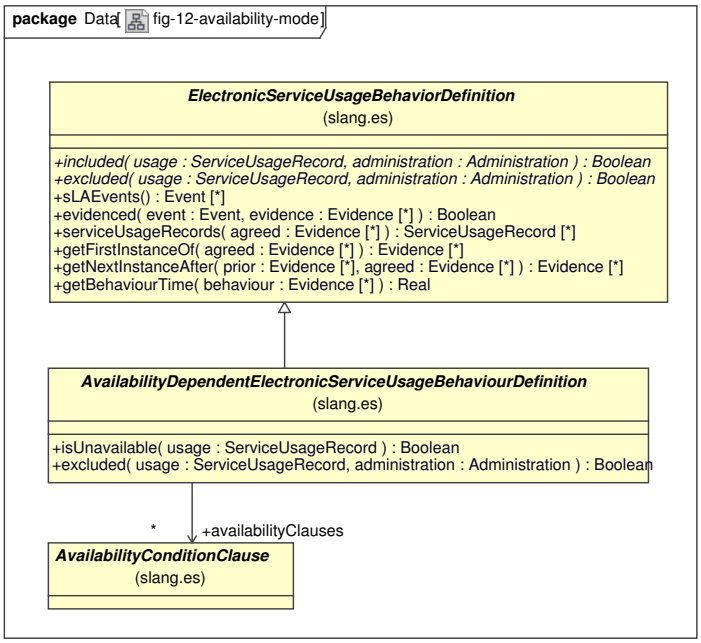


Figure 6.16: Electronic-service behaviours may be conditional on the state of availability of the service in some usage mode

If the client and provider of an electronic-service have agreed, via an exchange of reports, that the service is unavailable in some usage-mode, then it would be ludicrous for the client to be able to continue claiming reliability penalties supported by evidence of usages occurring in this mode. A straightforward way to encode this exception is to make the description of failure-modes dependent on the concurrent availability of the service (according to some availability clause). A class, `AvailabilityDependentElectronicServiceUsageBehaviourDefinition`, supporting this kind of behaviour definition is included in the language, and shown in Figure 6.16. It overrides the operation `excluded()` to state that a service-usage may not be regarded as part of this mode if the associated availability clauses is being violated when it occurs. This abstract class may be combined with `FailureModeDefinition`, for example, in an extension. An example of this is

given in the next chapter.

6.11 The SLAng language specification

SLAng is defined in a collection of source files in the format accepted by the UCL MDA tools. These sources may be compiled into a single XMI version 1.2 document. Ultimately, I would prefer this XMI document to be regarded as the definitive statement of SLAng, because XMI is a standard concrete syntax for EMOF and OCL, whereas the input format for the UCL MDA tools, despite its various advantages, is not. Unfortunately, tool support for compiling a model incrementally from sources in a mixture of formats does not currently exist.

Since SLAng is currently a fully abstract language, it will always need to be extended before it is used. At present, by far the easiest way to achieve this is to use the UCL MDA tools to create the sources for a new language. These new sources may use the import mechanisms that I have implemented in the UCL MDA tools to incorporate the sources defining the core SLAng language. Once a concrete language has been generated, it will be appropriate to compile it into an XMI file. As discussed in Chapter 4, concrete SLAs should then reference this file as the definitive specification of the language in which they are defined.

It is important for a potential user of SLAng to understand how the core language should be used, including the above methodological advice concerning the production of extensions. By far the most comprehensive discussion of issues surrounding the use of the language is this dissertation, which in due course will be made publicly available. Therefore, the SLAng language specification includes as informal advice associated with the `SLA` class, the recommendation that a user consult this document, together with the caveat that this document should not be considered to be definitive of the language.

6.12 Additional considerations in ASP SLAs

6.12.1 Payments and penalties

A prime requirement for SLAs is that they allow the provider of a service to charge for the use of their service. The only support that I have thus far provided for this is the `Compensation` class included in the domain model. Clearly, much more sophisticated support could be required, but this will not always be necessary. In the next chapter I provide an example of the use of throughput conditions to implement per-use charging.

6.12.2 Multiple penalties, gradated penalties, and interactions between conditions

Reliability clauses and failure-mode definitions in a SLAng SLA are associated in a many to many relationship. Several reliability clauses may apply concurrently, and violations and penalties are calculated for each independently. Failures may belong to multiple failure modes, so there is the potential that a single failure may result in the application of more than one penalty.

This may be what is desired in the agreement, and result in a gradated payment scheme where the aggregate amount of penalties paid is related to the number of violations occurring concurrently. However, it may be preferable to disregard penalties for more minor infractions when a more serious infraction is occurring. Support for this may be implemented by considering what violations of other

clauses have been calculated in the same or previous administrations for which violations of a particular clause are being calculated. A sophisticated example of this is provided in the next chapter.

An alternative approach for implementing gradated penalties for violations of service-behaviour restrictions is to relate the value of the penalty to the length of maximal violations observed.

6.12.3 Maintenance and scheduling

Maintenance of an electronic-service is not an event that can be observed by the client. An SLA defining conditions over an electronic service should therefore not offer guarantees on maintenance directly, as such guarantees will not be monitorable by the client. Since maintenance periods imply that other guarantees will not be met a better approach is to associate schedules with these other guarantees. This can be achieved by filtering the events that are considered to be relevant to a condition by overriding the `sLAEvents()` operation on the `ConditionClause` class. An example of this approach is provided in the next chapter.

6.12.4 Real-world behaviour and mutual monitorability

The class `MutuallyMonitorableSLA` includes the constraint that evidence submitted during any administration of the SLA must only depend on events that can be observed by both the client and the provider of any service being administered. SLA authors defining extensions to SLAng to express conditions over real-world events must respect this constraint. If they do not, but honestly model the witnesses to the event types that they define, then a failure to specify monitorable conditions will be revealed if the SLA is tested. This once again indicates the strength of the model denotational pattern in supporting the definition of good SLAs.

6.13 Language specification overview

The following sections provide class diagrams summarising the classes and relationships in the current version of the SLAng language. First the generic syntax and semantic classes are presented. These classes are generic in the sense that they are independent of the particular type of any services being constrained in an SLA.

Subsequently the syntactic and semantic types supporting the definition of conditions over electronic services are presented.

Finally I summarise the relationship between syntactic and semantic types. Directed relationships from a syntactic type to a semantic type tends to represent a denotational relationship. A relationship from a semantic type to a syntactic type represents a reference to some clause in a concrete SLA in some piece of evidence related to the behaviour of a service.

6.13.1 Generic syntax

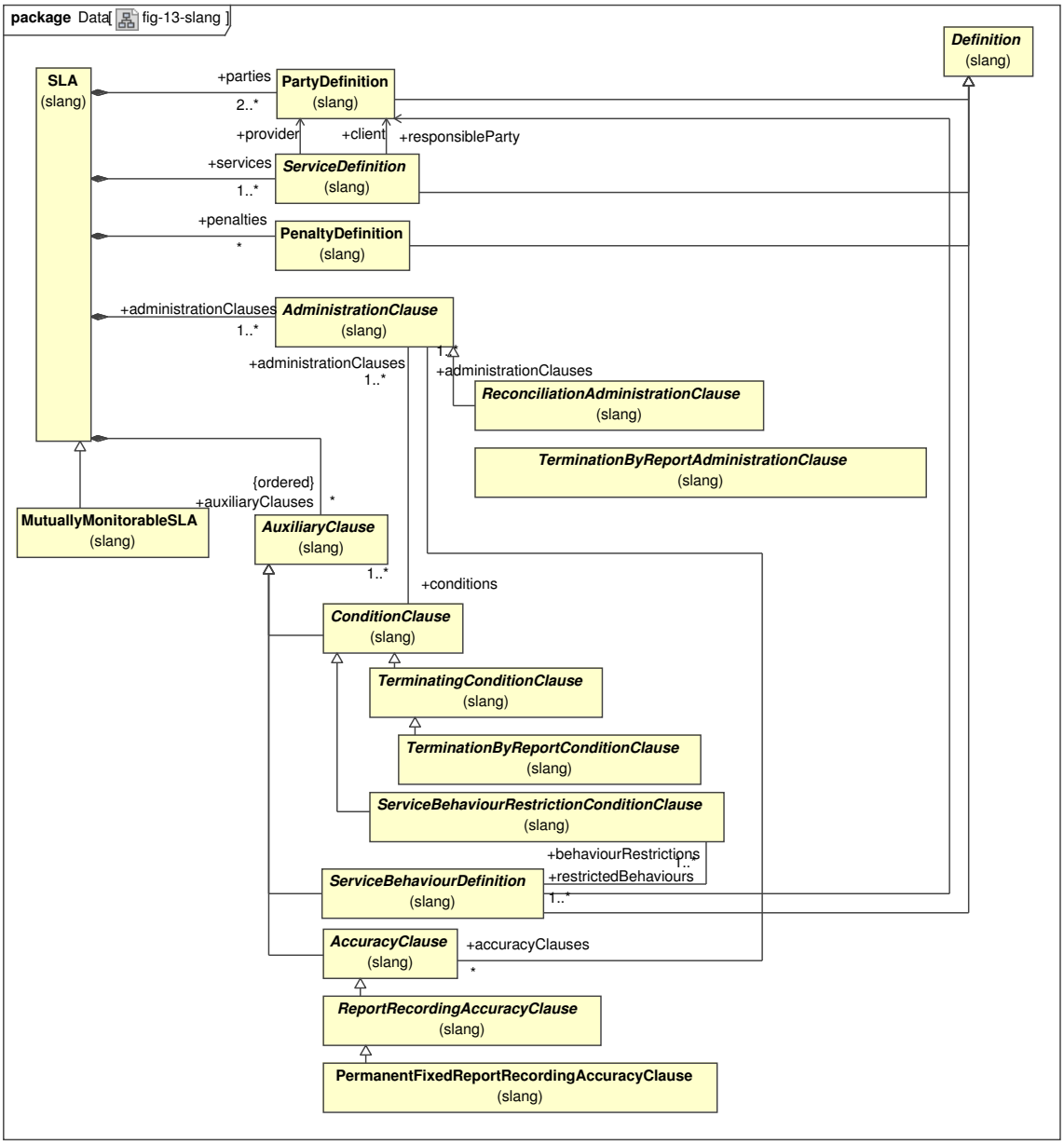


Figure 6.17: Syntactic elements supporting the specification of SLAs, but independent of service type, in the SLAng language specification

6.13.2 Generic semantics

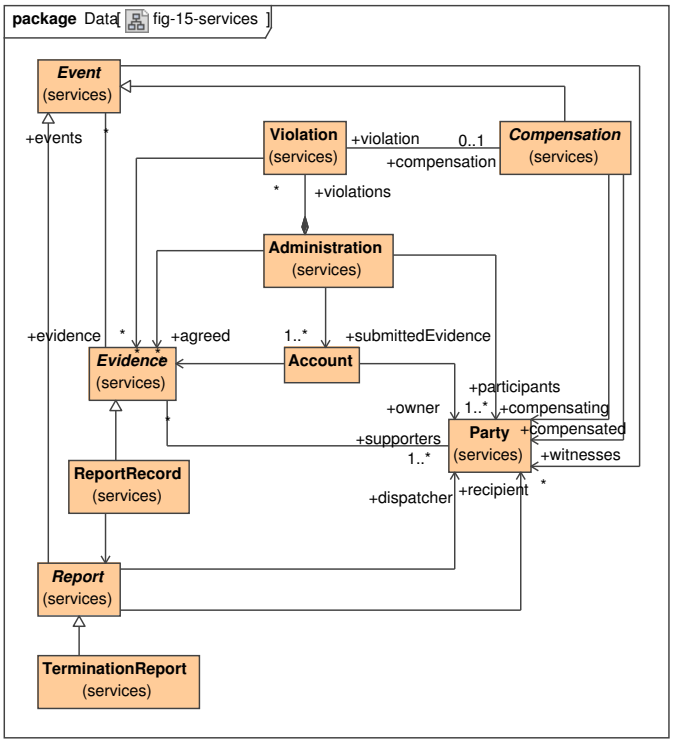


Figure 6.18: Semantic elements descriptive of SLA relationships independent of the types of service of which conditions are expressed, in the SLAng language specification

6.13.3 Electronic-service syntax

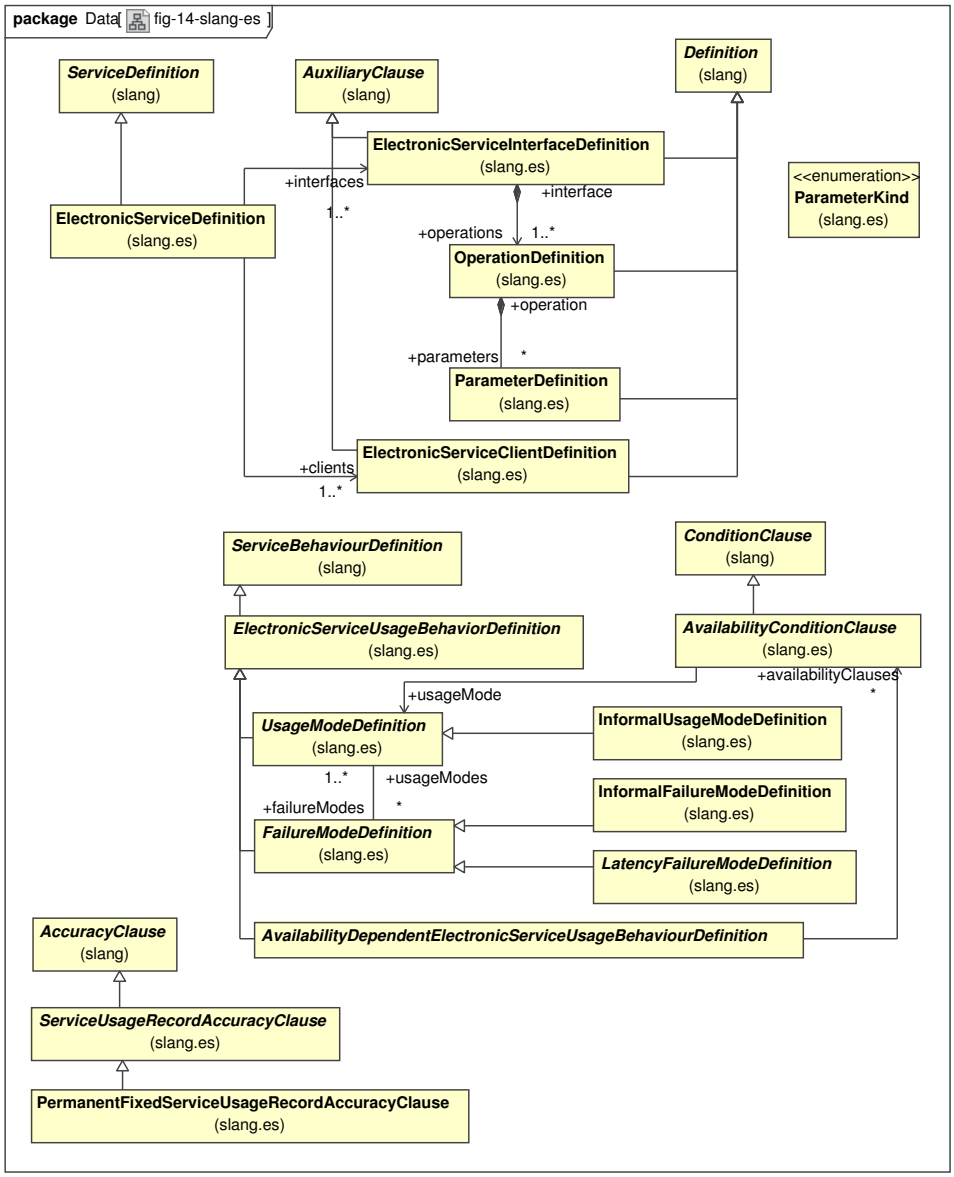


Figure 6.19: Syntactic elements supporting the specification of SLAs for electronic services, in the SLAng language specification

6.13.4 Electronic-service semantics

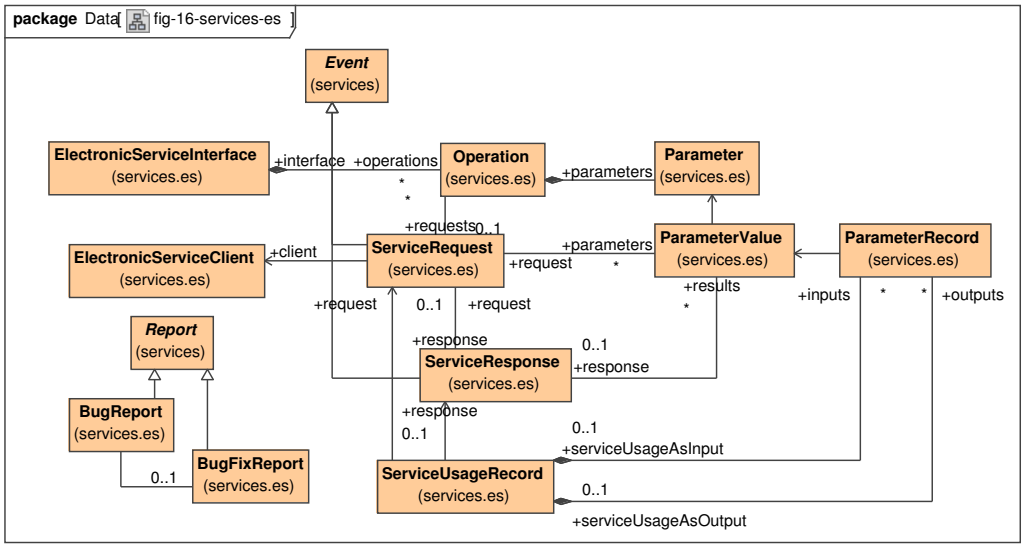


Figure 6.20: Semantic elements descriptive of electronic services in the SLAng language specification

6.13.5 Relationships between syntactic and semantic elements

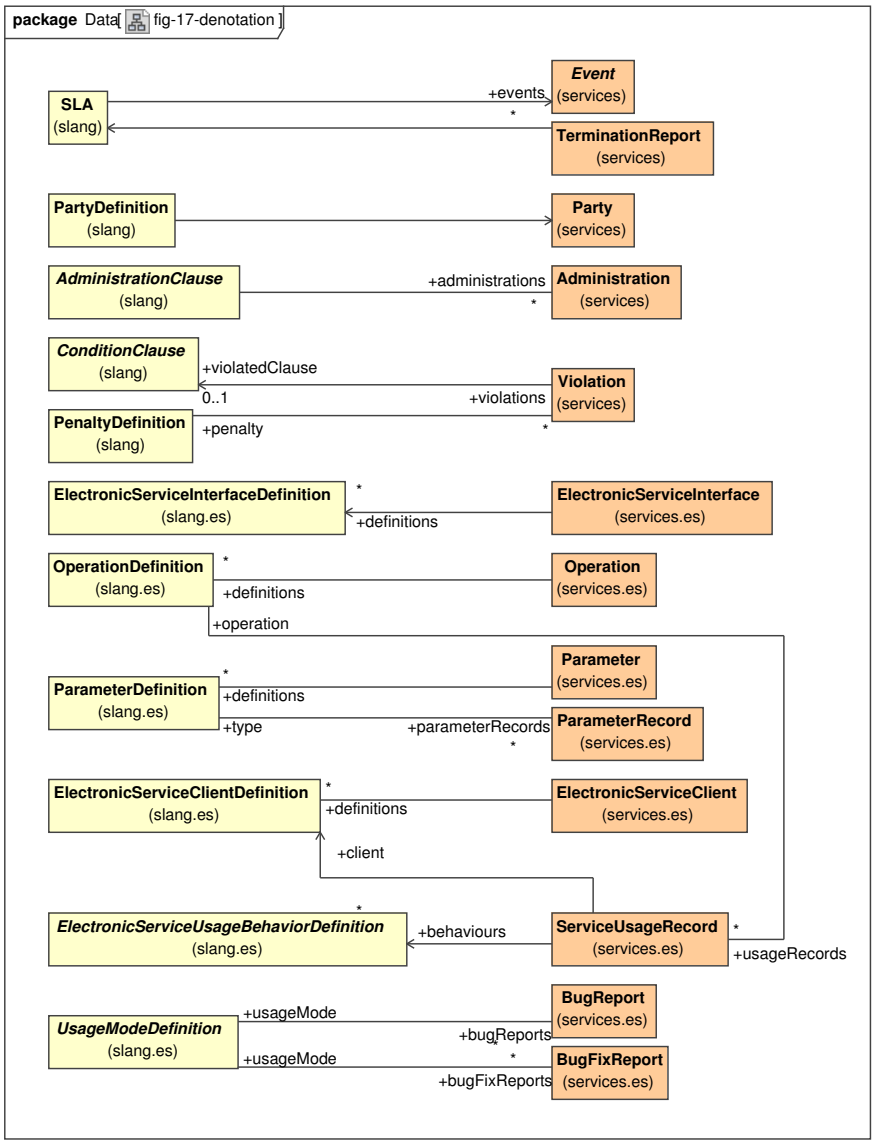


Figure 6.21: Relationships between syntactic and semantic elements in the SLAng language specification

6.14 Summary

In this chapter I have described the design of the SLAng language, which is an abstract, extensible language for ASP SLAs. The full language specification is available online [121] and documented in Appendix E, which also includes the definitions of extensions to the language described in the following chapter. I presented a thorough description of the syntactic and domain models included in the core language specification, and gave examples of OCL constraints relating elements in these models to define the semantics of SLAng in according to the model-denotational approach. I described how abstract classes and side-effect-free operations are used in the specification to assist users in defining the extensions necessary to support the specification of a concrete SLA.

In the next chapter I describe a case-study in the use of the language, thereby providing examples

of both extensions to the language, and concrete SLA statements expressed using the language.

Chapter 7

Case-study: the eMaterials project

In the previous chapter I have described the design of an abstract, extensible, domain-specific language, SLAng, providing support for the definition of mutually-monitorable SLAs with conditions defined in relation to electronic services.

In this chapter I present a case-study involving the use of SLAng to define SLAs in a realistic application-service scenario. This exercise provides the foundation for a validation of SLAng against its requirements, a comparison of SLAng with alternative languages for ASP SLAs, and a discussion of the evolution of the language, all of which are covered in Chapter 8.

This chapter also contributes an initial method for integrating SLAs into an existing service-provision scenario, and a demonstration of that method applied to develop some SLAs.

The case-study scenario chosen is a service developed by the Computer Science department at University College London as part of a now completed research project, eMaterials. The service, which I refer to in aggregate as the polymorph-search service, is provided to the Chemistry department to aid the chemists in performing computational analysis of chemical structures. The service relies on infrastructure services provided by other parties, including network services and the provision of computing nodes for a computational grid. It also involves the outsourcing of a graph-plotting service to Southampton University.

In the remainder of the chapter I first introduce the case-study method, then describe the case-study, resulting in two complete SLA examples.

7.1 Case-study method

The case study presented here investigated the use of SLAng as the basis for the definition of a number of SLAs. According to the terminology of [147], it was a *single-case study*, where the *unit of analysis* was a single service-provision scenario. The principal research question being addressed was whether the SLAng language can be used, with extensions, to define SLAs that would be satisfactory to the various stakeholders in a realistic ASP scenario. The main conclusion of the case-study, was is positive with respect to this question, and I *generalise* from this to suggest that SLAng would be appropriate for SLAs for other ASP scenarios, based on the observation that the chosen scenario does not seem to possess any special qualities, other than being of its nature an ASP scenario, that make it amenable to the use of SLAng. Validation of this conclusion is provided by establishing that the SLAs produced were indeed suitable to the scenario. This was achieved by evaluating the case-study SLAs according to the

requirements developed in Chapter 2. This evaluation is provided in Chapter 8.

The case-study was also intended to be *descriptive*, as the account of it given here illustrates the steps necessary to produce extensions to the SLAng language to support the statement of concrete SLAs. Furthermore, it had an *exploratory* component, in that it cast light on the following issues:

- the ease and expense necessary to use SLAng as a basis for defining SLAs;
- the analysis activities required to determine the appropriate design and parameters for the case-study SLAs;
- non-fundamental expressivity requirements that could lead to useful extensions to SLAng to increase its expected adequacy to future applications;
- stakeholder views on the need and uses of SLAs;
- process issues relating to the development of SLAs for a particular scenario.

I describe the case-study method chosen in detail in this section.

In choosing a scenario to study, I sought to match the following criteria, based on my assumptions regarding the applications and benefits of SLAs, and the focus of this work on SLAs for application-service provisioning:

- the scenario should be of practical or commercial interest to some parties other than myself;
- multiple financially independent parties should be involved;
- communication between the parties will be in part mediated by electronic services. Electronic services may also implement some functionality in the scenario;
- plausible requirements that may be satisfied through the use of SLAs should exist.

Evaluating SLAng in the context of pre-existing service scenarios is analogous to introducing a new technology to a software project late in the development process, or retrofitting a deployed system. The new technology has the chance to meet existing requirements in the scenario, or improve the degree to which existing requirements are met. However, the scenario has not been designed with the new technology in mind. Introducing the new technology may result in derived requirements that are not compatible with decisions that have already been made.

A practical approach to introducing a new technology into an existing development is as follows: first, an understanding of the state of development of the scenarios must be obtained. This will include an understanding of who the stakeholders in the scenario are, and what fundamental requirements the service-provisioning scenario is intended to meet for them; second, on the basis of the understanding of the requirements developed in the first step, requirements specifically relevant to the new technology are considered; third, a plan for the introduction of the new technology should be made, aiming to avoid modifying the existing scenario significantly, which would imply redevelopment costs; fourth, this attempt is evaluated in terms of the additional advantages provided by the technology with respect to the

requirements that it is capable of satisfying, and any associated costs or disadvantages that introducing the new technology might imply. In the fifth stage, recommendations may be made for redeveloping the initial scenario to better accommodate the derived requirements that introducing the new technology implies. If the third stage is highly successful, the fifth may not be required.

The case-study I present here followed this pattern. The steps taken, and the information elicited or produced at each stage are shown in Figure 7.1. In the following subsections I describe the steps in more detail, as they relate specifically to the introduction of SLAs into a scenario.

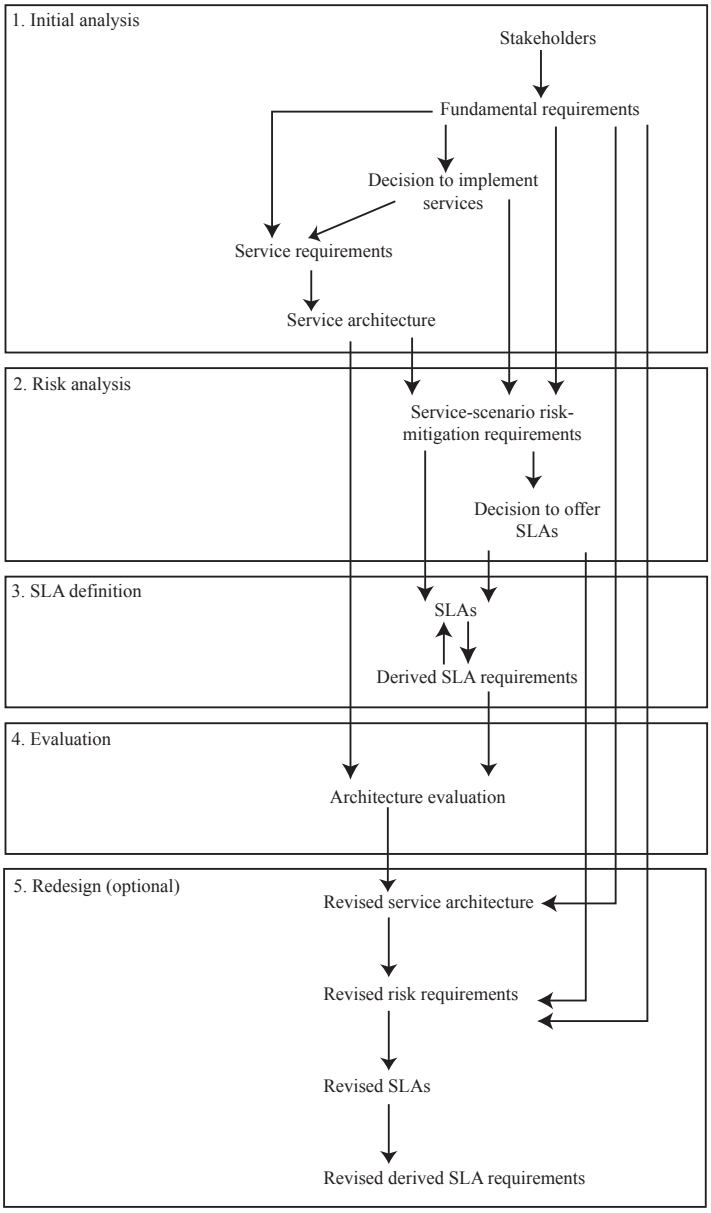


Figure 7.1: Case study phases, and the information gathered in each. Arrows indicate derivation relationships between the information, with the target of an arrow derived in some part from the source.

7.1.1 Initial analysis

In the first stage of the case-study, a view of the existing scenario was developed. This includes a requirements analysis and model of the existing service architecture.

An analysis of the fundamental requirements underlying the scenario is needed because risk-mitigation requirements will depend on the objectives of the various parties engaging in the scenario. It is the assumption of this work that SLAs serve the purpose of mitigating financial risk in a service-provision scenario, so acquiring an understanding of these requirements was the principle focus of the analysis part of the case-study.

Moreover, when considering a redesign of the scenario to render it more amenable to the use of SLAs, choices can only be justified on the basis that they meet the fundamental requirements to a superior degree than the original design.

The requirements analysis performed in the case-study consisted of a stakeholder analysis, followed by the development of a summary of the requirements for the scenario from the perspective of each stakeholder.

To accommodate the limited resources that case-study participants had to devote to the case study, the model of the scenario and its requirements was be derived primarily from consultation with a Principal Stakeholder (PS), in this case the computer-science researcher with responsibility for coordinating the polymorph-search service. Requirements elicitation concentrated on high-level requirements, and requirements were not investigated in much detail. To compensate for this lack of rigour, the principle stakeholder was consulted throughout the case-study to ensure that important assumptions or requirements that may have initially been overlooked were discovered as the case-study developed.

The model of the scenario developed in this stage consists of:

- an overview description of the scenario;
- a UML deployment diagram depicting the service architecture;
- a list of stakeholders in the scenario and their fundamental requirements;
- use-case descriptions for services in the scenario.

7.1.2 Risk analysis

In the second stage of the case-study, the risks to the stakeholders of using the existing service architecture were modelled. Risks imply a requirement to mitigate them, and my assumption is that SLAs are an appropriate mechanism for mitigating some such risks. Therefore the purpose of the risk analysis was to identify specific requirements that could be addressed using SLAs.

The risk analysis was guided by considering each step in the service-provision use-cases and how undesirable outcomes of these steps could result in harm to the participants in the scenario.

Analysis concluded when an agreement with the PS with respect to the risk requirements was reached.

7.1.3 SLA design and definition

Having identified the risks faced by the participants in the scenario, the next step in the case-study was to attempt to produce SLAs to mitigate the risks, without recommending any change to the processes or the deployment of services in the scenario. This effort naturally broke down into two steps, which I call SLA design, and SLA definition.

In the SLA-design stage, a system of SLAs was first proposed, which I hoped would mitigate to some extent all of the risks for which SLAs are an appropriate risk-mitigation mechanism in the scenario. At this point the only details decided for each SLA were the origins of the events in the service deployment in relation to which conditions of the SLAs would be specified. The choice of SLAs in the system was not arbitrary, but was informed by the desire to provide a system of SLAs that was as monitorable as possible. Anticipating the need for latency constraints in the SLAs, and identifying service interactions similar to the three-party scenario considered in Chapter 5, it was possible to argue for a particular system of SLAs as being optimal.

The next step in the SLA design was to decide how the individual SLAs would contribute to mitigating the risks of the scenario participants. This was achieved by considering, for each SLA, each risk in the scenario, and whether the SLA could contribute to its mitigation. For a particular risk one or more particular conditions for an SLA were proposed that would mitigate the risk. For example, the risk that the simulation may not complete in a timely fashion led to the proposal of conditions for an SLA between Chemistry and IS, resulting in penalties paid by IS to Chemistry in the event of slow or faulty simulation completion. At this point the conditions were proposed in an abstract and informal manner, identifying the intent of each SLA without specifying parameter values, or precise meanings for the conditions.

Entering into the proposed SLAs would be undesirable for the case-study participants if the SLAs posed additional risks to the scenario participants that they did not also mitigate. Therefore, I extended the risk analysis into the SLA design phase of the case-study. When designing the conditions for the SLAs, I attempted to identify and then mitigate any new risks. This led to a somewhat iterative approach to SLA design, where introducing a new condition in one SLA could imply the need for a complementary condition in another. I maintained a list of all risks in the scenario separately from the risk-analysis and design documentation, which now constitutes Appendix B.3. Each risk is cross-referenced both to the point in the analysis or design documentation at which it is identified, and to the proposals for SLA clauses to mitigate them, so it is clear what new risks are introduced and how they are dealt with. At this point in the case-study, some risks were discounted as not being suitable for mitigation by SLAs, such as security risks for which existing hardware and software risk-mitigation approaches exist.

The product of this phase in the case-study was a description of a system of SLAs, and for each SLA in the system, a list of the clauses required in each SLA, specified informally, but related to the risks they are intended to mitigate. The system required for the eMaterials scenario consisted of five SLAs. At this point I also listed the details of the electronic-service interfaces in relation to which the SLAs will need to be specified, to the extent that these details could practically be determined. At this point in the case-study, progress was hindered by a lack of documentation for certain communications protocols used by the electronic-services in the scenario. As I result, I elected to exclude one of the required SLAs from further consideration.

In the SLA-definition stage I attempted to convert the informal, abstract descriptions of the conditions for the remaining SLAs into fully-specified, formal conditions expressed using SLAng and extensions to SLAng.

The creation of these SLAs, and the extensions on which they depend, required two further related efforts, one of analysis, the other of design. The analysis effort was to determine parameter values for the SLAs, for parameters for such clauses as latency and reliability conditions, and penalty definitions. The design effort was to produce SLAng extensions, capturing the structure and the meaning of the required clauses, and subsequently concrete SLA statements relying on these extensions to implement the SLAs. These efforts were related because the structure of the extensions determines what parameters values are required.

As discussed below, I discovered that the four remaining SLAs could be grouped into two pairs, with the SLAs in each pair having identical structure. This result, combined with difficulties experienced in obtaining meaningful parameter values for the SLAs led to a narrowing of the case-study focus to the development of two fully-formalised SLAs.

For each condition of the two SLAs in question, I performed the following steps iteratively:

1. I considered how it could be implemented using SLAng, or extensions to SLAng;
2. I used the structure of SLAng and its putative extensions to guide the production of questions for the PS designed to elicit parameter values for the conditions, or refinements of the proposed extensions required to capture peculiarities of the required agreements;
3. I put the questions to the PS, and make decisions regarding the design of the SLA based on the answers received.

7.1.4 Evaluation

Having completed and documented the initial SLAs, the results were submitted to the PS for comment. The proposed SLAs were assessed according to the extent to which they mitigated the risks identified in the scenario, and the practical implications of using the SLAs, in particular in terms of monitoring responsibilities.

At this point, it was possible to address the principal research objective of the case-study, as a set of SLAs had been proposed. I also critically reviewed the case-study process, in the hope of proposing further improvements to this nascent method for SLA development.

7.1.5 Redesign

Certain problems encountered during the initial analysis, design and definition of a system of SLAs for the eMaterials scenario could more profitably be addressed by modifying the scenario, rather than deciding that the SLAs themselves are inadequate. In a short discussion of these problems, I consider modifications to the scenario to better accommodate the use of SLA technology.

From the point of view of the principal research question in this case-study, this stage was unnecessary as it had already have been demonstrated that SLAng can be used to specify appropriate SLAs. However, the stage was interesting from an exploratory perspective.

7.2 The eMaterials case-study

The eMaterials project, now complete, funded a collaboration between UCL grid-computing researchers in the Department of Computer Science (CS) and the UCL Chemistry Department to investigate the

computational prediction of organic crystal structures from chemical diagrams.

The problem is relevant to the discovery of new drugs, but is computationally demanding in general. A large number of molecular packings must be considered for each compound, the thermodynamic likelihood of each being indicated by a calculation of the lattice energy. Physical properties of likely crystals must then be estimated. The problem amounts to search in a large space, coupled with sophisticated analysis of the candidates.

Prior to the eMaterials project, the chemists would execute this search using two Fortran programs, `MOLPAK` and `DMAREL` and a combination of manual and batch control, on a 4 CPU Silicon Graphics server. A typical search would take between one and four months to complete [22].

Within UCL, the Information Services (IS) division is a support group that manages computational and network resources for the administrative departments and the student population. It also administers the inter-departmental network. Individual departments may also have independent groups fulfilling the same role for the academic staff, and this is the case for the Chemistry and Computer Science departments.

The eMaterials project funded the creation and administration of a computational grid, controlled by researchers in the UCL Computer Science (CS) department, but consisting of nodes maintained by IS. The aim was to support the analysis activities of the chemists while providing the opportunity to research grid engineering for computer-science.

7.2.1 SLAs in the eMaterials scenario

This scenario represents a potentially interesting use of SLAs. The eMaterials project is now complete, and without a centralised source of funding the various participants in the simulation infrastructure must consider how their costs are to be covered. In practice, this may be achieved either by acquiring additional research grants in return for the promise of future scientific and technical advances, or alternatively the various departments and the university administration may consider the services involved valuable enough to fund out of overhead costs. The possibility of commercialising the service has also been suggested. If this were to occur, there would be a definite need to consider SLAs.

For the purpose of this case-study, I adopt the assumption that the parties remain financially independent, and that the principle benefit of the infrastructure is to the chemists. They must therefore pay for it from funding into minerals research. The various service providers must recuperate their costs by charging for their services. In return, their clients may expect them to provide quality-of-service guarantees.

7.3 Service architecture

The polymorph-search service architecture makes use of a number of technologies, which I now briefly review:

7.3.1 `MOLPAK` and `DMAREL`

The jobs implementing the computational simulation performed by the polymorph-search service are implemented by the Fortran application programs `MOLPAK` and `DMAREL`. `MOLPAK` currently supports 38 different packing types that can each generate up to 200 candidate packings. The physical properties

of these packings are then calculated by DMAREL.

Computationally these applications are independent of each other. Subject to resource availability, the 38 packing types can also be evaluated in parallel. This enables the problem to be solved by nodes in a computational grid without the use of shared memory, and with low bandwidth connections. The implementation of the applications are also independent and conversion between input and output formats is required. Individual jobs typically take between 5 minutes and an hour to complete.

A typical workflow for the application is shown in Figure 7.2.

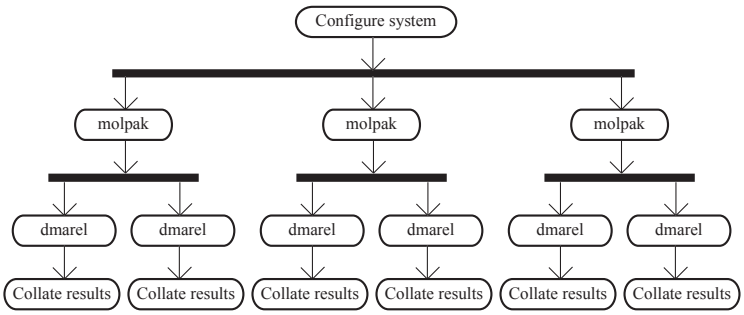


Figure 7.2: Workflow in the polymorph-search service

7.3.2 Condor and Polynet

The eMaterials project obtained cooperation from the overall UCL administration and in particular IS to make use of surplus computational resources in the collegiate network of cluster rooms (rooms full of standard PC equipment used by undergraduate students for coursework and general computing).

This resulted in the deployment of a grid cluster of consisting of around 1000 IS-managed machines. The cluster software and access to these resources is managed by CS research staff and all management software is installed on CS-managed servers. Job submission naturally originates in the Chemistry department.

Job scheduling within the cluster is managed by Condor, a grid middleware [14]. Condor manages a set of computational resources according to some scheduling policy. It maintains one or more job queues, to which serial or parallel job can be submitted. A Condor management node will then, according to its policy, allocate computational resources to these jobs.

Initially, access to the UCL Condor-grid resources was provided using a thick-client called Polynet, coded in Java, which integrated directly with Condor. The simulation workflow was essentially fixed in the implementation of the client, although a degree of parameterisation was possible. Overall simulations that previously took 1 to 4 months could now be completed in a matter of hours.

7.3.3 ActiveBPEL Workbench

The Condor/Polynet solution was criticised as limiting the control that the chemists have over their own simulation procedures, which was not the case when the simulations were manually supervised.

In response, grid research in CS has focussed on providing a workflow editing and enactment environment based on the language BPEL [102], and the open-source technologies Eclipse (a development environment) [19] and ActiveBPEL (a workflow enactment engine) [1]. This environment is called

the ActiveBPEL Workbench [20].

In this environment, scientific workflows can be written in Java with the help of a graphical editor. These may then be deployed into an ActiveBPEL container. ActiveBPEL exposes management interfaces to deployed workflows as web-services and web-pages. Custom ports for these services may also be specified, the invocation of which correspond to the initiation of new parallel activities within the process, proceeding from 'receive' actions. Because web-service invocations can also be specified within workflows, it is clear that these facilities allows the hierarchical composition of workflow-based web-services, as well as the orchestration of traditional web-services. ActiveBPEL is deployed as a Java servlet [127] in the Apache Tomcat application server [5], and relies on the Apache Axis library for its implementation of web-services [7].

To enable the coordination of the UCL Grid it was necessary to integrate ActiveBPEL with Condor. It was not deemed desirable to modify ActiveBPEL to this end, as one of the reasons for selecting BPEL for orchestration was that industry would tend to produce better workflow enactment engines than the research community. Clearly requiring custom extensions to the workflow engine mitigates against this.

At the time of implementation of the polymorph-search service, Condor did not implement web-services directly, although this functionality has subsequently been contributed to the Condor project by researchers in the UCL CS department. Therefore, the decision was made to wrap Condor in an interface defined by GridSAM, an open-source project with the goal of providing standard interfaces for distributed resource managers, discussed in more detail below.

Combined with GridSAM, these technologies enable the scripting of simulations for execution on grids. UCL call these simulation orchestrators 'meta-schedulers'.

It is not yet clear the extent to which this effort has been successful because of the unwillingness of chemists to obtain the skills required to script BPEL workflows. In an effort to demonstrate the efficacy of the approach, CS has implemented a meta-scheduler for the eMaterials workflow, which was previously coordinated by the Polynet client. The preferred method for chemists to conduct simulations is now via a website interface to this meta-scheduler. This webpage, implemented using a combination of static pages and Java servlets hosted in Tomcat, is called the Polymorph Search Webclient.

7.3.4 GridSAM and JSDL

GridSAM is an open-source job submission and monitoring program [30]. It's main control interface is implemented as a web-service. The project is funded by the UK Open Middleware Infrastructure Institute (OMII) managed programme [98].

The aim of GridSAM is to provide a standard interface for the submission and monitoring of scientific processing jobs. The GridSAM implementation also provides a degree of support for executing these jobs, and common deployment tasks associated with them, in particular the gathering (or 'staging') of related data resources. Much of this functionality is implemented by grid middleware such as Condor.

A job specification is expressed using the Job-Submission Description Language (JSDL) [99], an

XML dialect and standard of the Open Grid Forum (OGF). The job specification includes details of the procedure to execute the job (typically by executing a program within an operating system environment), the run-time resources required, and the locations of any pertinent data (which may include the executable artifacts for the job).

The job specification does not include any scheduling information for the job. Having received a job specification, the GridSAM service coordinates with a Computational Resource Management System (CRMS), typically grid middleware such as Condor. GridSAM, via CRMS-specific extensions, essentially translates any information included in the JSDL specifications it receives into configuration files and actions appropriate to the underlying CRMS.

JSDL has some of the characteristics of an SLA language, in that it describes resource requirements which are expected to impact on the quality-of-service delivered by the grid. A comparison of the expressive capabilities of JSDL to SLAng is provided in the next chapter.

7.3.5 The `plotws` service

The collation of simulation results by the polymorph-search service involves the production of a summary graph as a bitmapped image that can then be retrieved from the Polymorph Search Webclient. The production of this image is achieved using the plotws service, a stateless web-service hosted by Southampton University.

Collation of results occurs after the completion of each DMAREL job, so that the progress of a simulation can be viewed incrementally.

7.3.6 Service deployment

I now describe the deployment of the polymorph-search service, depicted in Figure 7.3.

The Polymorph Search Webclient provides a web-service interface that may be used by the chemists to initiate a simulation. This is achieved in two steps: first, large parameter data is provided by HTTP upload. Second, a simulation is initiated by an HTTP request containing some additional small parameters. The node hosting the webclient is called trout1. This node also contains a Condor submit queue.

When a simulation is started, the webclient interfaces with a meta-scheduler on a second node, trout5. This node then orchestrates the processing of MOLPAK and DMAREL jobs by submitting JSDL specifications to a GridSAM instance on the same node as the Polymorph Search Webclient, trout1. This GridSAM instance converts these specifications into Condor submissions and submits them via commands issued to the Condor submit daemon which maintains a queue of submissions.

Periodically, a Condor controller process, running on a third node managed by CS, polls the submit queue resident on trout1. This controller also maintains information about free resources in the cluster, provided by periodic status updates delivered by the Condor processes running on the cluster nodes. If suitable resources are available, the controller notifies the cluster node and the queue, effectively assigning the node to the queue. The queue process then manages the staging of any configuration or data files to the cluster node. Note that because the queue process is on the same node as the Polymorph

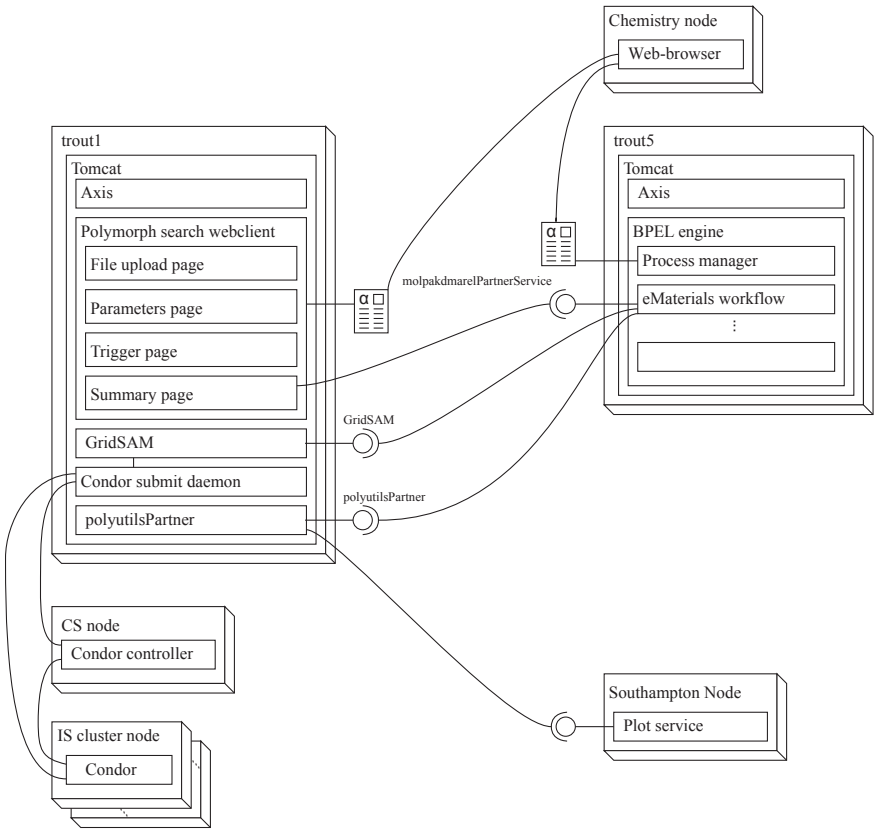


Figure 7.3: Service infrastructure in the e-Materials case-study

`Search Webclient` (`trout1`), any files uploaded by the chemist when configuring the simulation will be available for staging.

`MOLPAK` and `DMAREL` are both serial jobs so do not benefit from any concurrency within the grid. The logic of concurrent execution is entirely captured by the workflow depicted in Figure 7.2. Both tasks are computationally demanding so are run exclusively, one node per job. The vast majority of nodes in the cluster have only a single CPU.

The cluster nodes process jobs that have been assigned to them. When complete, they notify the queue that assigned the job, and stage any results files back to the node on which the queue is deployed. They also notify the `Condor controller` that they have become available.

Periodically the meta-scheduler executing on `trout5` will poll the `GridSAM` service to determine the status of some job. `GridSAM` in turn acquires this information from the `Condor submit daemon`. Once a job is found to have completed, the meta-scheduler may submit further jobs. Following `MOLPAK` jobs, `DMAREL` jobs are scheduled.

When a `DMAREL` job completes, the meta-scheduler coordinates the production of a results summary webpage. This is effected by invoking the `polyutilsPartner` web-service deployed on `trout1`. In the course of its operation the `polyutilsPartner` service invokes the `plotws` web-service hosted by Southampton university. This produces an image containing a graph on which is plotted a set of data points, each generated by a completed `DMAREL` job. The simulation is complete when all `DMAREL` jobs have completed and the final summary generated.

The various processes and nodes mentioned above are located in networks, and communicate via networks controlled by the different parties in the scenario. Figure 7.4 depicts the locations of the nodes in networks and the communication pathways used between the nodes.

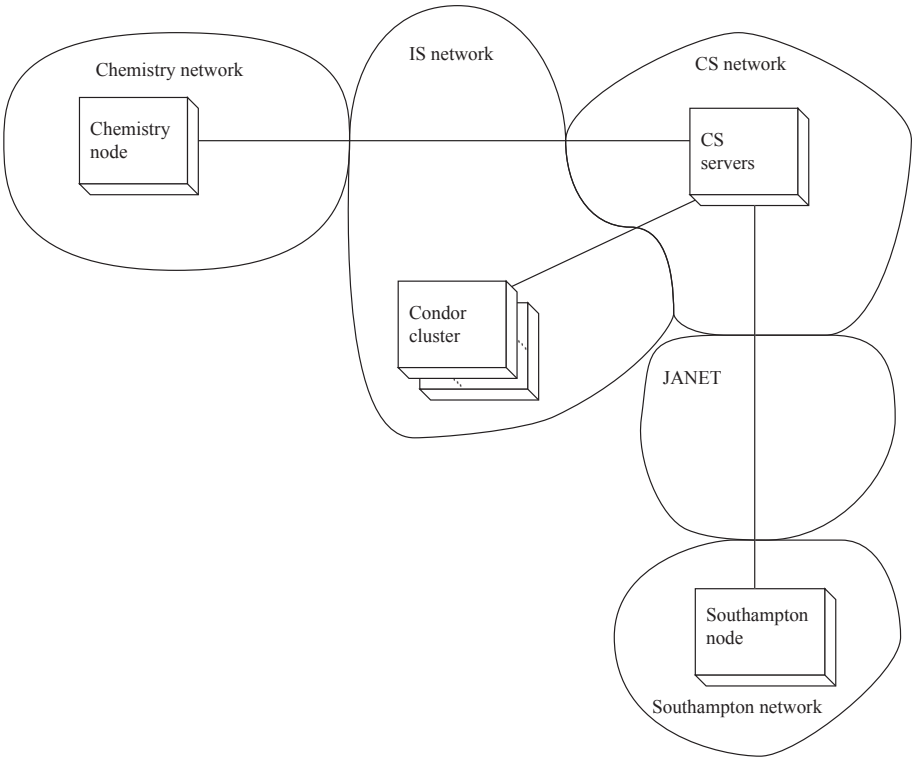


Figure 7.4: The location of nodes within networks in the eMaterials scenario

7.4 Stakeholders and fundamental requirements

The stakeholders in this scenario are the following, listed along with their fundamental requirements for the scenario:

Chemist: wishes to obtain the results of a specific simulation in a timely and correct manner. I assume that the Chemistry Department takes responsibility for the behaviour of chemists in this scenario, and will enter into SLAs on their behalf.

Computer scientist: wishes to enable the execution of the simulation, while covering the costs of contributing network and processing resources to the simulation. Computer scientists wish to be involved with the undertaking of simulations by scientists from other disciplines in order to formulate theories and provide solutions to better enable this activity in the future. I assume the Computer Science Department takes responsibility for the behaviour of computer scientists in this scenario, and will enter into SLAs on their behalf.

Information Services division: wishes to recuperate costs from services provided to academic departments within UCL. These services include making surplus computational resources available and providing network services connecting departments within UCL.

ISP: any internet service providers connecting UCL and Southampton wish to generate revenue by providing network connectivity. In fact, in this scenario there is a single ISP, JANET [41].

Southampton University: wishes to generate revenue by charging for the use of a graph-plotting service.

7.5 Use-case and risk analysis

7.5.1 Use-cases in the scenario

The scenario consists of three main use-cases. These are:

1. Conducting a simulation – in which a chemist triggers a simulation to be executed on the grid;

2. Administering grid nodes – in which a computer scientists accesses and configures individual grid nodes provided by IS;

3. Receiving grid status notifications – in which grid nodes provided by IS periodically and autonomously report their status to the `Condor Manager` residing on a node controlled by CS.

Each of these use-cases involves interaction between at least two of the scenario participants, who I am assuming are financially independent. Therefore, there is a risk that a fault will occur which is the responsibility of one party, but which harms a second party. It is the objective of this case-study to produce SLAs to mitigate these kinds of risks.

Interdependencies exists between the use-cases. If a computer scientist is unable to configure cluster nodes, or if those nodes fail to report their availability for processing in a timely or accurate fashion, it is unlikely that the computer scientist would be able to provide any guarantees concerning a simulation that they coordinate.

Despite the importance of all three use-cases, in this case study I have opted to consider only the first use-case, that of conducting a simulation. The reason for this is that to write SLAs, or to include SLA terms related to the second and third use-cases would require an understanding of the detailed communication protocols that `Condor` uses for administration and status reporting. Unfortunately this is impractical, because these protocols are not documented, and the effort required to reverse-engineer `Condor` places such an exercise outside the scope of this work. This issue is discussed further in the case-study conclusions.

7.5.2 Use-case 1: conduct a simulation

Appendix B.1 contains a thorough risk-analysis of the primary use-case for the eMaterials scenario, namely conducting a simulation, which I summarise here.

This use-case involves all of the scenario participants. To successfully complete a simulation the following steps must complete correctly and in a timely manner:

1. A chemist collects the parameters, and input and output files for the simulation.

2. The chemist uploads parameters and input files using the `Polymorph Search Webclient`.

3. The chemist triggers the computation and the webclient acknowledges the start of the computation by displaying an initial status.

4. The webclient triggers the `eMaterials workflow` installed in the BPEL engine on the workflow node.

5. The `eMaterials workflow` triggers the individual jobs by passing JSDL specifications to a `GridSAM service` instance installed on the submission node.

6. The `GridSAM service` translates the JSDL to a `Condor` submission file and places it in the `Condor` submit queue by signalling the `Condor submit daemon`.

7. The `Condor controller` node polls the `Condor submit daemon` on the submit node occasionally for information about the queue. When it discovers new submissions, it applies its grid scheduling policy to allocate grid nodes to processing these submissions, informing the submit daemon of the location of the allocated nodes, and each grid node of their allocation to the submission (effectively granting access control of the nodes to the daemon).

8. The `Condor submit daemon` contacts the allocated grid nodes for a submission, stages the parameters and input files to them and instructs them to begin processing.

9. The grid nodes process their individual jobs.

10. The grid nodes notify the submission node when they have completed their jobs. They then stage the results files back to the submission node.

11. The BPEL engine polls the `GridSAM` service for the status of jobs on the queue. The completion of a job may trigger the scheduling of additional jobs. Following `MOLPAK` jobs, `DMAREL` jobs are scheduled. This step includes the repetition of steps 5 – 10.

12. Each time a `DMAREL` job completes the workflow engine coordinates the production or update of the results website by invoking the `polyutilsPartner` web-service on the submission node.

13. As part of its operation the `polyutilsPartner` web-service invokes the `plotws` web-service in Southampton to prepare a scatter graph of the results.

14. The Chemist occasionally checks the results website. When the results are ready, the chemist may view the plot of the results and download result-data files.

In Appendix B.1 I use the use-case as a framework for itemising the risks to which the parties in the scenario are exposed. For each step, I attempt to consider, for each involved party, what could possibly go wrong and how that party will suffer. The risks that the parties suffer due to participation in the scenario may be summarised as follows:

The Chemistry department is the principal recipient of benefit in the scenario. The risks to chemists stem from the possibility that the simulation might not complete, might deliver incorrect results, or

that conducting a simulation will be hindered by usability issues affecting the `Polymorph Search Webclient`, which presents its functionality as a website. The Chemistry department also assumes some additional risks as a result of the need to interact with other parties. These include security risks, due to the need to accept into their own network traffic appearing to originate from within the CS network. Also, if Chemistry regards the results of its simulations to have any proprietary value, then they assume a risk related to the possibility that experimental data will be stolen when transmitted across networks, in particular the Internet. Finally, Chemistry, by depending on a service provided by one or more second parties, assumes a termination risk, based on the possibility that those parties may choose to render the service permanently unavailable at some point, resulting in reintegration costs for Chemistry.

The Information Services division of UCL initially assumes risks based on its interaction with other parties. These include the security risks of interacting with Chemistry and CS. IS also suffers a risk associated with allowing CS to install and execute software on their computational nodes. IS provides both network and cluster-node services, so assumes two risks due to the potential volume of legitimate service requests. IS is also exposed to the risk, when providing these services, that they will not be reimbursed for the costs involved. Since IS does not depend on the service being delivered, and the initial risk analysis does not assume the use of SLAs, IS at this stage could mitigate the latter risk by not providing the service. However, IS has a fundamental requirement to charge for provided services, so I assume that this is not an option.

The computer-science department similarly assumes security risks, risks related to resource exhaustion by legitimate service requests, and the risk that they will not be compensated for the resources they contribute to the performance of the simulation.

The ISP assumes security risks and resource exhaustion risks implicit in permitting interactions between CS and Southampton. The ISP must also find a way to charge for the use of their resources.

Southampton also assumes security and resource exhaustion risks providing the `plotws` service. Southampton also wishes to (at least) recuperate its costs from providing the service, so runs the risk that it will provide the service but then be unable to do so.

7.6 SLA design and risk analysis

7.6.1 A system of SLAs for the scenario

I now consider what SLAs are appropriate for mitigating the risks identified in the previous section. In principle, several different systems of SLAs might be satisfactory to the participants. However, I have elected to attempt to design a system offering the highest possible level of monitorability for the SLAs that it contains.

According to the results of the monitorability analysis described in Section 5.1.3, mutually-monitorable SLAs in a three-party service provision scenario, with a client, provider and network-service provider are the most monitorable SLAs achievable if latency conditions are required. Moreover, only a single configuration of such SLAs is *safe*, and that is where two SLAs are used, each between two parties that share a direct interface between their infrastructures.

The case-study scenario includes two similar sub-scenarios: the provision of the `Polymorph`

`Search Webclient` by CS to Chemistry across the IS network, and the provision of the `plotws` web-service by Southampton to CS across the Internet. In both cases latency conditions will be required in order to mitigate the enumerated risk **Chemistry-6**, identified in Appendix B.1, pg. 268, that the production of results will be delayed (see Appendix B.3).

These sub-scenarios differ from that considered in my original analysis in two respects: first, the eMaterials scenario contains five parties, rather than three, and the possible influence on monitorability of the two extra parties should not be neglected; and second, my original analysis assumed that the client and the provider of the service were nodes embedded in the network of the network-service provider. In the eMaterials scenario in contrast, all computational nodes are embedded within networks controlled by the same organisations that control the nodes.

I now argue that these differences make only a small practical difference to the monitorability result, allowing the reuse of this result to inform the choice of SLAs in the scenario.

Consider the interaction between Chemistry and CS across the network provided by IS: the two extra parties in the scenario are the ISP and Southampton. Clearly they are not respondents to (i.e. cannot observe) any of the events in the interaction directly, because they do not have a trusted platform within the Chemistry, CS, or IS network from which to gather data. Neither can they monitor the events indirectly by having them reported to them, since only Chemistry, CS and IS could do such reporting, and they are barred from doing so because they will necessarily have to enter SLAs concerning these events, and therefore have an interest, and therefore cannot report. An analogous argument holds for the interaction between CS and Southampton across the internet.

Concerning the differences in network configuration: service usages in both sub-scenarios still have end-to-end QoS requirements, such as latency requirements. However, requests and responses pass over two extra network segments, owned by the client and the service provider respectively, in addition to the segment owned by the network provider. Faults causing delays can occur at any point in the infrastructure, including in the client or service-provider's network. Clearly, it would not be safe for the network provider to insure correct behaviour in these sections (without reciprocal guarantees from the other parties, which would be pointless).

However, the interfaces to these peripheral network sections may be regarded as being similar to nodes embedded in the central network. The provider of the electronic service in each case can guarantee the performance of their own network segment, and hence provide good service to the network provider at their mutual interface. The network provider can therefore guarantee good service at the interface to the client network, but not beyond. QoS guarantees can only be provided for the client as far as the boundary to the client's network. However, if the client manages their network correctly, this will allow them to guarantee the end-to-end QoS of the service, so this is adequate.

Consequently I decided that an appropriate system of SLAs for the scenario must include SLAs to govern service provision at the interfaces between the Chemistry, IS, and CS networks, the Internet and Southampton's network for the two interactions already discussed. These SLAs will be mutually monitorable, which will be the best degree of monitorability obtainable for these sub-scenarios, without

the introduction into the scenario of additional parties, or trusted monitoring solutions (the theory of which is not well understood, at the time of writing).

The remaining interaction in the scenario is direct interaction between CS and IS to pass information and commands to and from cluster nodes. In this case a single mutually monitorable SLA at the network boundary between CS and IS suffices, since there is no intervening network operated by a third party. Note that a more monitorable SLA is not possible because no other party can monitor these events, and no other location for delivering guarantees would be appropriate since the parties can neither monitor, nor guarantee the behaviour of the service within their peers' networks.

Figure 7.5 shows the set of SLAs chosen for the scenario located at the network boundaries at which events pertinent to their conditions occur.

These recommendations result in two SLAs involving both IS and CS. Clearly these SLAs could potentially be combined into a single SLA, although I have not at this stage chosen to do so. I discuss the consequences of this in the case-study conclusions.

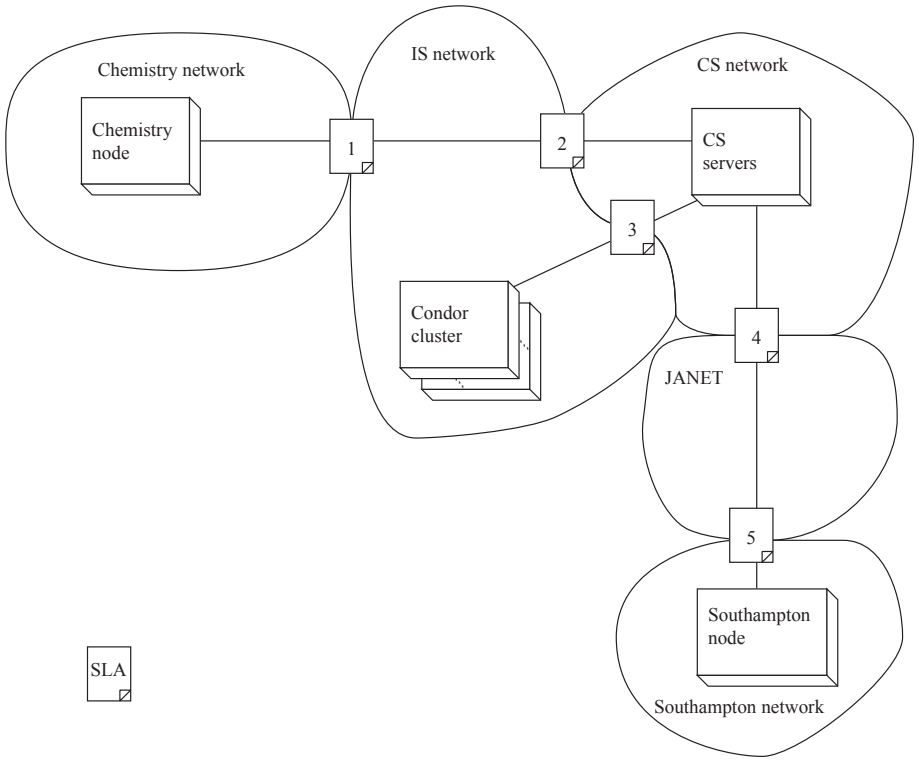


Figure 7.5: SLAs for the eMaterials scenario, located at network boundaries where events occur, to which they are pertinent

7.6.2 Individual SLA design

In Appendix B.2 I describe in an informal and abstract manner the design of each SLA proposed in the case-study. This is achieved by considering, for each SLA, each risk in the scenario, and whether the SLA can contribute to its mitigation. The appendix also list the details of the electronic-service interfaces in relation to which the SLAs will need to be specified.

Entering into these SLAs poses additional risks to the participants in the SLAs, such as the risk

that IS will need to pay a penalty due to faulty service behaviour caused by CS (a 'safety' risk, in the terminology of Chapter 5). Therefore, I extend the risk analysis into this phase of the case-study. All new risks are included in Appendix B.3, and cross referenced to the proposals for SLA clauses to mitigate them, or the decision to omit them as requirements for the system of SLAs.

I defer summary of designs of the individual SLAs to the next section, where they can be presented together with the description of the SLA definitions.

At this point in the development of the case-study, three issues affecting the continued progress of the case-study became clear.

The first was that it was appropriate to rule out a number of risks identified in the scenario as being inappropriate for mitigation by SLA. These include security risks of any kind. Some security risks are more appropriate for mitigation by implementation mechanisms in the service itself. Others, such as the identified need to keep simulation data confidential, an SLA could help to mitigate. However, this is a wide field of investigation that I have already ruled outside the scope of this work, and discuss further in the final chapter.

Having ruled out these risks, it was possible to design conditions in the system of SLAs that mitigated all remaining risks and all risks assumed by the parties as a consequence of the use of SLAs in the scenario.

Second, the design process revealed that the clauses required by SLA 2 are identical in kind to those required by SLA 1, and this is also the case for SLAs 4 and 5. In retrospect, this was predictable based on the fact that these pairs of SLAs each represent the provision of the same service across two network boundaries. It is therefore to be expected that (in the absence of other considerations for the parties, such as bulk service provisioning) the SLAs involving the final consumer will only differ from those involving the original service provider by accommodating slightly longer delays and more failures, thereby accommodating the behaviour of the network.

The third important issue that became apparent at this stage was that although it was possible to understand in abstract how SLA 3 should mitigate the risks to CS of nodes operated by IS performing inadequately (in aggregate), it was not going to be practical to fully specify this SLA, since to do so would require the reverse-engineering of the protocols by which the `Condor queue` and `Condor manager` communicate with nodes in the grid, an effort beyond the scope of this work. This was the same problem that led me to avoid considering the second and third use-cases in the scenario, which deal with this type of communication exclusively.

In the next section I therefore consider how the abstract set of clauses required for each SLA, apart from SLA 3, may be implemented using extensions to SLAng.

7.7 SLA definition

To this point in the case-study I have detailed (in Appendix B.2) the conditions required for each SLA in the scenario for the system to be effective overall in mitigating the risks to the participating party. I have also identified the details of the service interfaces. This is sufficient information to commence development of SLAs formalised using SLAng, and extensions to SLAng.

At this point in the case-study, a problem with the methodology became apparent. Although the approach of proposing conditions based on the requirement to mitigate the scenario risks appeared to have correctly identified the types of conditions required, both I and the PS struggled to make the conditions fully concrete. This was due to a lack of complete knowledge concerning at least the following aspects of the scenario:

- the costs involved in providing the service;
- the magnitude of the financial harm borne by the parties as a result of a violation of one of the SLA conditions;
- what degree of degradation of the service is genuinely problematic to the chemists using the service;
- what the usual load and capacity of the `plotws` service is.

Uncertainty on issues of this nature is understandable since the scenario thus far has not been operating on a genuinely commercial basis, so no serious effort has been made to investigate these issues.

Unfortunately, the effort required to investigate and draw meaningful conclusions in relation to these issues places them outside the scope of this work. As I discuss in the concluding chapter of this dissertation, these issues are not entirely unrelated to the provision of language support for SLAs, since understanding the economic effect of an SLA requires an understanding of its precise meaning, as determined by the language in which it is specified. However, the issues also touch on broader research topics such as capacity planning and the financial management of services which require greater consideration than can be afforded here.

Fortunately, however, it is still possible to demonstrate the use of SLAng to produce realistic SLAs. A reasonable approach to designing language extensions is to anticipate their possible reuse in some future SLAs, and therefore implement them in a parameterisable way. Because the syntactic structure and meaning of the conditions required for the SLAs should be largely independent of particular parameter values it is possible to argue that demonstrating that some SLA using these conditions can be written implies the possibility of expressing an *appropriate*, or even ideal, SLA using the conditions, assuming the conditions themselves are appropriate and appropriate parameter values can be specified. In this section I describe the definition of two such example SLAs, and support the relevant assumptions by arguing that the conditions implemented are appropriate based on the scenario requirements, and by attempting to choose plausible parameter values, even if optimal values cannot be determined.

In the previous section I ruled out the production of SLA 3 on the basis of a lack of knowledge concerning the protocols implemented by `Condor`. I also observed that SLAs 1 and 2, and 4 and 5 are identical in the conditions that they require, at least at an abstract level. I now rule out the production of SLAs 2 and 5, on the basis that they will be structurally identical to SLAs 1 and 4, or close enough that elaborating the differences would not demonstrate anything useful about the expressive power of SLAng. Therefore, the only differences these SLAs will have with SLAs 1 and 4 are parameter values,

and since these values involve a large component of guesswork, it will not be meaningful to elaborate these SLAs.

Therefore, in the following sections I describe concrete SLA definitions for SLAs 1 and 4, each consisting of a set of language extensions plus a concrete SLA document.

7.7.1 SLA 1: Provision of the `Polymorph Search Webclient` by IS to Chemistry

In this section I describe the language extensions supporting, and the concrete statement implementing, SLA 1. All language-extension types described in this chapter have equivalents in Appendix E, which is an automatically-documented language specification containing the complete SLAng language, plus a combination of the types required to express SLAs 1 and 4. SLA 1, expressed in HUTN syntax, constitutes Appendix C.

The clauses required in SLA 1 are as follows:

1. latency conditions on the setup operations of the webclient;
2. an availability condition relating to the service;
3. reliability conditions on the setup operation of the webclient;
4. latency condition on the simulation-invocation operation of the webclient;
5. reliability condition on the simulation-invocation operation of the webclient;
6. latency condition on the amount of time taken for simulation results to become available;
7. reliability condition on results retrieval operations;
8. throughput conditions on all operations;
9. a payment scheme, charging Chemistry for use of the service;
10. a termination penalty for IS terminating the service;
11. a limit on the rate at which simulations can be started;
12. a guarantee concerning the performance of `MOLPAK` and `DMAREL` from Chemistry;
13. a termination penalty for Chemistry terminating the agreement.

In addition, the SLA must describe:

- how the SLA will be administered;
- what standards of accuracy the parties must adhere to when gathering evidence for the calculation of violations.

The various clauses implementing these conditions have a certain amount of interdependence. I now describe the overall structure of SLA 1, followed by the definition of these conditions. I discuss conditions with fewer dependencies first, in an attempt at clarity. In the discussion below I have not provided a highly-detailed description of how the extensions override abstract classes or operations of the SLAng language core. However, I have included base-classes from SLAng in the illustrative class-diagrams where appropriate, and it is notable that the overwhelming majority of the syntactic extension types benefit from support from base-classes defined in SLAng. This is also apparent from inspection of the extensions in Appendix E.

Structure of SLA 1

SLA 1 gets its overall structure from the definition of the SLA type in SLAng, without the need for any extensions to this type. Throughout the following discussion of the SLA, I cross-reference my discussion of the SLA clauses to the listing of the SLA in Appendix C. In the following subsection, I excerpt from the SLA to show how the general throughput constraint for SLA 1 is specified. However, in subsequent sections I do not reproduce parts of the SLA in the main body of the text. The overall structure of SLA 1 is given below.[1]

- A list of parties involved in the SLA. SLA 1 identifies Chemistry and IS, beginning at line 24, pg. 292.
- A list of services over which the SLA places conditions. Naturally, SLA 1 only describes the `Polymorph Search Webclient`, beginning at line 35, pg. 292. An electronic-service description includes the following:
  - the identification of the provider and client parties (by reference to parties previously defined in the SLA);
  - a list of electronic-service interfaces constituting the service. In this case, the website interface to the `Polymorph Search Webclient` is described starting at line 45, pg. 293. An interface definition describes all operations of the interface, and all parameters for those operations;
  - a list of deployed client software, permitted to access the service, line 254, pg. 297;
  - a list of behaviours for the service, starting at line 263, pg. 297. These are any behaviours that may be relevant to the definition of condition clauses later in the SLA. SLA 1 defines numerous behaviours of the webclient, many of which are also dependent on whether violations of certain conditions are discovered when the SLA is administered.
- A list of penalty definitions, that may be referred to when defining the penalties for violating conditions defined later in the SLA, starting line 838, pg. 309.
- A list of administration clauses. Each administration clause defined rules for when and how violations of the SLA are to be calculated. An administration clause defines at least the following:

[1] Readers may find it helpful to read the following sections using a PDF viewer, as these cross-references are implemented as hyperlinks for convenient browsing

  - a list of conditions that should be checked against the submitted evidence to calculate violations. For example, those specified starting at line 974, pg. 312;
  - a set of accuracy clauses, governing the standards by which various types of evidence should be collected for the purposes of administration (see line 951, pg. 311, for example).

  SLA 1 defines two administration clauses, both pertaining to the same set of conditions, as discussed further below;

- A list of auxiliary clauses. Auxiliary clauses are of a diverse set of types, but share the common characteristic that they may be reused in multiple settings in the SLA, and so are notionally subcomponents of the top-level SLA element, rather than any other clause. Condition clauses are all examples of auxiliary clauses as they may be reused in multiple administrative clauses. Auxiliary clauses are listed starting at line 1176, pg. 316.

General throughput condition

This condition-definition implements condition 8, listed on pg. 193.

The condition that is least dependent on any other is an input-throughput condition obliging the client to limit their attempts to use the service, whether successful or not.

This is implemented using a service-behaviour restriction associated with an informal usage-mode description, which is in turn associated with all of the operations of the service. SLAng requires no extension to describe the usage mode. The service interface, including all operations, is defined in SLA 1 as part of the definition of the overall polymorph search service, which in fact only consists of access to this single interface. Definition of the service begins on line 35, pg. 292. The interface is defined between line 45, pg. 293 and line 251, pg. 297. Here I reproduce the first several lines of the service definition, showing the references made to the definitions of the client and provider parties, and the start of the service-interface definition, including the definition of an operation returning a static webpage:

```
31 es::ElectronicServiceDefinition[polymorph](
32 ”The provision of the Polymorph Search Webclient by IS to Chemistry”
33 ) {

35     provider = PartyDefinition[uclis]

37     client = PartyDefinition[chemistry]

39     interfaces = {

41         es::ElectronicServiceInterfaceDefinition[polymorph](
42             ”HTTP interface to the Polymorph Search Webclient”) {

44             owner = PartyDefinition[uclis]

46             operations = {

48                 es::OperationDefinition[static1](
49                     ”http://sse.cs.ucl.ac.uk/omii−bpel/polymorph/index.htm”) {

51                     parameters = {
```

```
53                          es::ParameterDefinition(
54                              ”HTTP response status code”, OUT),
55                          es::ParameterDefinition(
56                              ”HTTP response message body”, OUT)
57                      }
58                  },
```

Note that a full definition of the structure of the service-interface is required in the SLA, as details of operation parameters, for example, imply monitoring obligations for the SLA participants. The usage mode is specified informally at line 265, pg. 297. I also repeat it here:

```
263 es::InformalUsageModeDefinition[anyUsage](
264     ”Request of any page”) {

266     operations = {

268         es::OperationDefinition[static1],
269         es::OperationDefinition[static2],
270         es::OperationDefinition[static3],
271         es::OperationDefinition[submit],
272         ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke],
273         es::OperationDefinition[results],
274         es::OperationDefinition[results1],
275         es::OperationDefinition[results2],
276         es::OperationDefinition[results3],
277         es::OperationDefinition[results4],
278         es::OperationDefinition[results5],
279         es::OperationDefinition[results6],
280         es::OperationDefinition[results7],
281         es::OperationDefinition[results8]
282     }
283 }
```

Although informal, it can be seen that the usage-mode is explicitly associated with the definitions of each of the operations in the interface, including the first static page, as listed above.

I have frequently opted to use informal behaviour descriptions for behaviours described in SLA 1 and SLA 4. A service-usage record submitted as evidence in and administration of one of these SLAs should indicate membership of any of these behaviours if the usage conforms to the natural-language description of the behaviour specified in the SLA.

The alternative to this approach would be to formalise each behaviour using extensions to the SLA, such that the semantics of the agreement obliged the parties to label usage records with behaviours whenever the usage could be shown to have some formally specified characteristics. At least for those behaviours related to the functional reliability of the service, by which I mean the relationship between input and output parameter values, this would mean the production of a complete formal description of the service, an impractical effort. In the case of these SLAs, it is also probably an unnecessary one: the required functional behaviour of the service is tolerably well understood by the SLA participants (Chemistry may seek compensation from IS for failures that IS does not understand well; however, IS will then seek to obtain compensation from CS under the terms of SLA 2, and so can rely on CS's expertise concerning failures when negotiating with Chemistry, to some extent); also, the SLAs are mutually-

monitorable, so a degree of reconciliation between the parties regarding the perceived behaviour of the service can be expected to occur when the SLAs are administered.

Decisions need to be made concerning the applicability of the service behaviour condition clause, with respect to when it applies, what the width of its sliding time window should be, how much concurrency of behaviours (in this case service requests) should be allowed, and what penalties should be associated with the condition.

In consultation with the PS it was decided that: (i) the throughput clause should apply continuously throughout the duration of the agreement; (ii) the window size and concurrency would be fixed and no penalty would be levied against Chemistry for a violation; and (iii) IS would not be liable to pay penalties for failures or delays while Chemistry was simultaneously violating the throughput constraint. The third provision was encoded into the SLA by introducing the new concept of a violation-dependent electronic-service behaviour mode, discussed further below.

Figure 7.6 shows the set of behaviour-restriction condition clauses derived from `Service-BehaviourRestrictionConditionClause` in support of SLA 1. The three concrete types of clause are all permanent, with a fixed window and fixed maximum number of occurrences permitted, and associate penalties with maximal violations, meaning that any interval in which any overlapping window represents a violation is treated as a single violation. The clauses differ in the penalties they apply – either no penalty, a fixed penalty, or a stepped penalty dependent on the duration of the maximal violation. The clause applying no penalty is used to implement the general throughput condition. SLA 1 specifies that Chemistry may not make service requests (to the operations jointly or severally) at a rate of more than 20 per 10 second period, at line 1030, pg. 313. The listing of this condition is as follows:

```
1028 ::combined::slang::—
       PermanentFixedWindowFixedOccurrencesNoPenaltyMaximalServiceBehaviourRestriction —
       ConditionClause[throughput]() {

1030     restrictedBehaviours = {

1032         es::InformalUsageModeDefinition[anyUsage]
1033     }

1035     maxOccurrences = 20;

1037     window = ::types::Duration(10, S)
1038 },
```

Simulation throughput and per-use charging

These condition-definitions implement conditions 9 and 11 listed on pg. 193.

The PS suggested that a latency guarantee, described below, could be met with good reliability, provided the chemist undertook to submit no more than a single simulation per day.

The submission of a simulation is indicated by a successful request to the simulation-invocation operation of the service. This behaviour therefore needs restricting in a service-behaviour restriction clause.

In Section 6.9.2 I described capabilities in SLAng for describing usage and failure modes of elec-

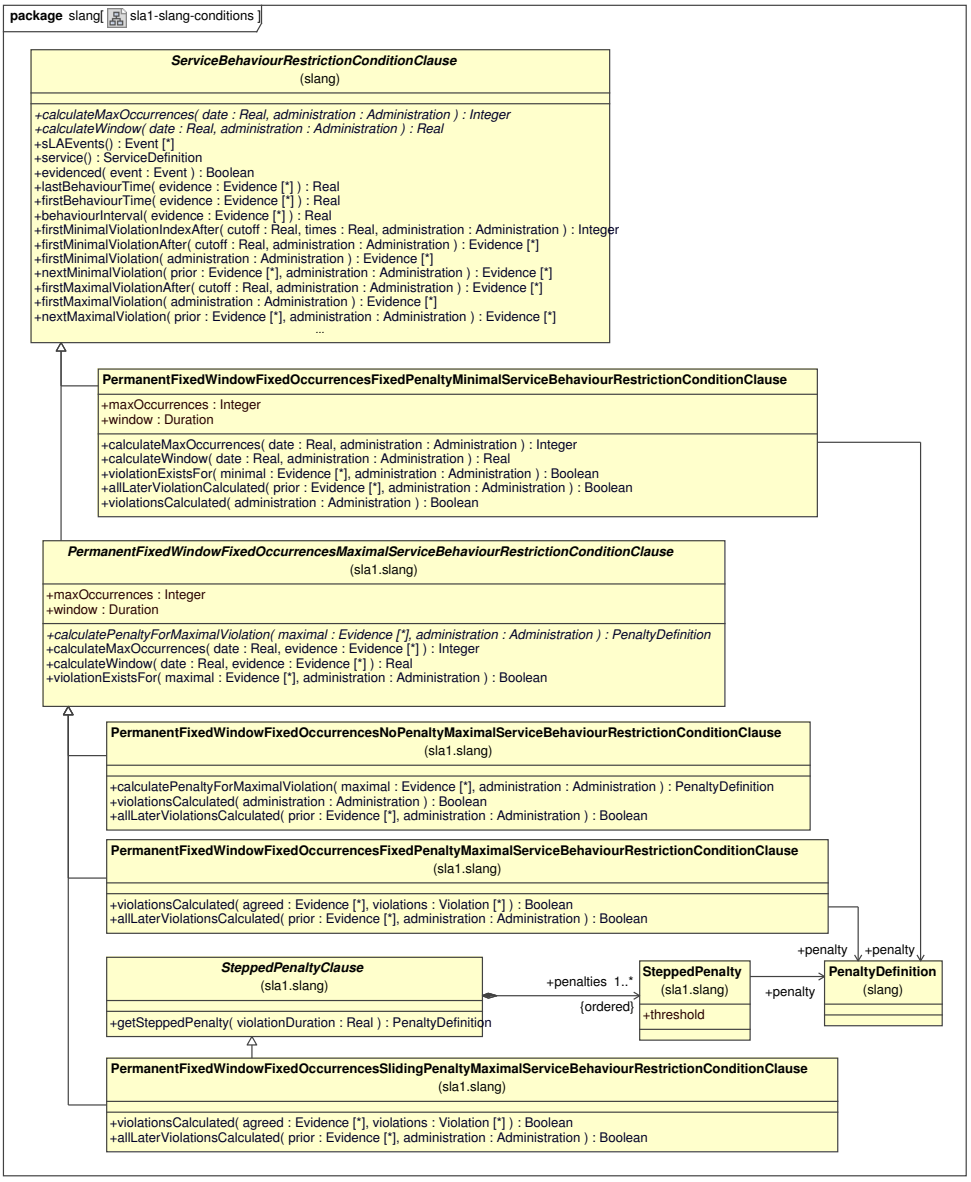


Figure 7.6: Service-behaviour-restriction conditions extended for SLA 1

tronic services. Neither of these concepts capture the notion of a successful request. Therefore, in an extension to the language it is necessary to add syntax for describing a type of behaviour that only successful service-usages can exhibit, a success mode. Support for this is shown in Figure 7.7.

The simulation-throughput condition is implemented by restricting a behaviour described by an informal success-mode definition (line 670, pg. 306) associated with the invocation operation (line 99, pg.294), to a single occurrence in a 24 hour window. Again, no penalty is levied for a violation, but the chemists cannot receive penalties for slow simulation execution if they violate this clause. The condition is included at line 1117, pg. 315.

The use of throughput conditions on successful operation also suggests a scheme by which charging can be implemented. A minimal behaviour-restriction condition clause associates a violation with the smallest set of behaviour exceeding its occurrences limit. Such a clause with an occurrences limit of 0

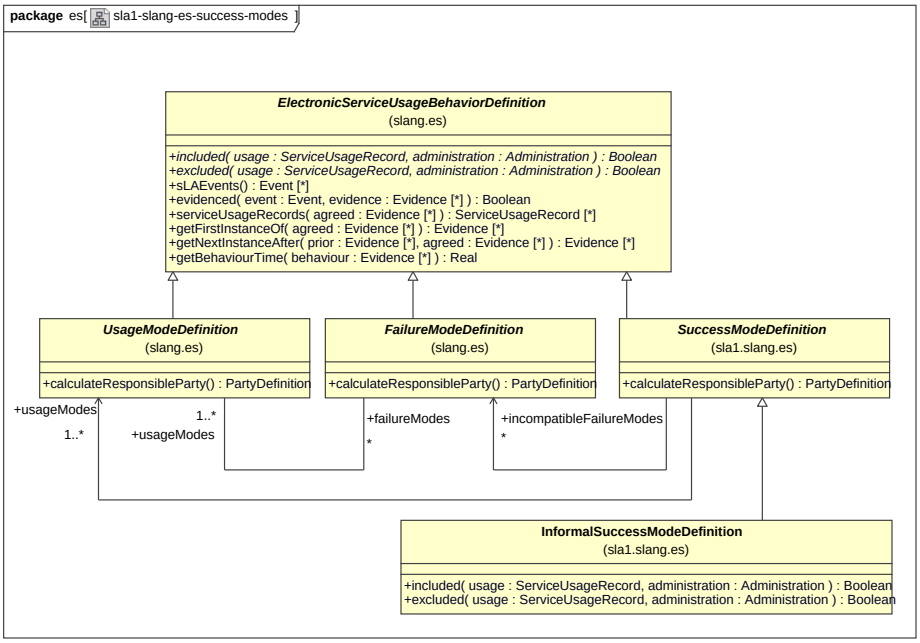


Figure 7.7: Success-mode types for SLA 1, enabling the definition of positive outcomes, supporting the definition of the simulation-throughput condition

can be used to associate a violation with each instance of a behaviour. Associated with a success mode, this may be used to implement per-usage charging. The condition clause is shown in Figure 7.6. The condition implementing charging is defined in SLA 1 at line 1130, pg. 315

Since the service in question is being provided in Great Britain in 2007, penalties associated with violations will be paid in Pounds Sterling. Figure 7.8 shows extensions to both the syntactic and domain models to describe such penalties. A Pounds-Sterling payment penalty definition requires the payment of some amount of money within some deadline of the associated violation having been calculated. In SLA 1 the amounts and deadlines are all fixed. All penalties are defined starting at line 838, pg. 309. The specified magnitudes of the penalties are fictitious.

General latency and reliability conditions

These condition-definitions implement all conditions related to latency and reliability of groups of operations in SLA 1, with the exception of those relating to the latency and reliability of the production of simulation results. I have elected to apply the same conditions for all operation groups, so the conditions implemented are 1, 3, 4, 5, 7 and 8, listed on pg. 193.

The Chemist requires penalties to be associated with delays and failures of the `Polymorph Search Webclient` because such events impact their ability to schedule simulations and retrieve their results.

Three types of failure mode are relevant. First, the service may be slow to a degree that it poses a nuisance but is still usable. Second, the service may exhibit delays that are so long that a chemist should not reasonably wait for the operations to complete, so should regard them as failures. Third, the service may produce faulty results.

In Section 6.9.2 I described an abstract type for latency failure-modes. To support a condition

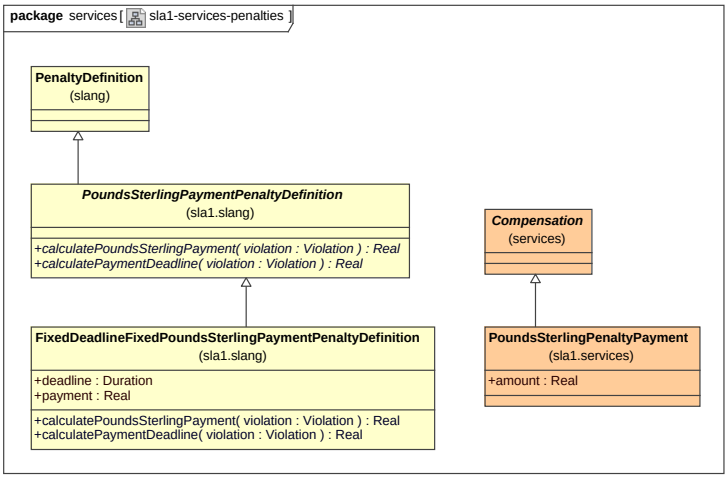


Figure 7.8: Syntactic and semantic elements supporting the definition of penalties requiring the payment of a sum of money in Pounds Sterling

restricting nuisance delays this must be extended to specify the maximum tolerable latency, and any circumstances under which usages taking longer than this threshold should not be considered an instance of the failure mode.

One such type of exception has already been described. If a usage simultaneously contributes to a violation of the chemist's input-throughput condition, then it should not be regarded as a latency failure, even if it takes too long. Generalising from this, I have provided a language extension introducing the notion of a violation-dependent electronic-service-usage failure mode.

The other exception that must be admitted depends on a condition related to availability of the service discussed below. If the chemist and IS have agreed that the service is unavailable by an exchange of bug reports, and the problem has not yet been fixed (indicated by an exchange of bug-fix reports), then the chemist cannot expect to receive compensation for failures. Consequently a notion of availability-dependent failure modes is also required.

Following the advice of Nielsen on web-usability [75], it was decided that delays over 10 seconds should be regarded as inconvenient. Delays of over 30 seconds should be regarded as intolerable. It was decided there was no reason for this threshold to vary. Therefore a fixed-latency, availability-dependent, violation-dependent, failure-mode definition, combined with a permanent service-behaviour restriction is adequate to penalise nuisance delays. This is defined in SLA 1 starting at line 319, pg. 298.

Irritating behaviour is tolerable in small amounts. Therefore I decided that penalties for irritating slowness should be levied if more than two requests within a minute take more than 10 seconds. Penalties for such an event should be very small, but increase somewhat if the same behaviour is observed for more than ten minutes. To capture this increasing penalty, I implemented the notion of a stepped-penalty condition-clause, whereby the applicable penalty varied as the duration of the violation exceeded certain thresholds. This clause is shown in Figure 7.6. This condition is defined at line 1073, pg. 314.

A stronger penalty should be applied if the majority of service requests take more than 30 seconds to complete. This would be the case if 10 delays of at least this duration were observed within an interval

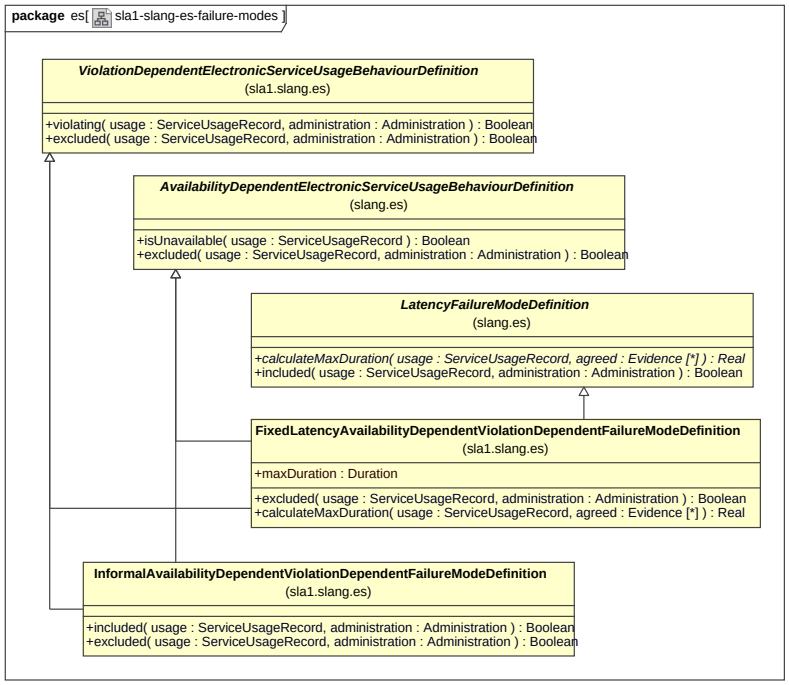


Figure 7.9: Latency, and informal, functional, failure-mode types for SLA 1

of 10 minutes.

Clearly after experiencing 10 delays of 30 seconds each within a 10 minute period, 5 minutes of the Chemist's time will have been wasted, they can conclude that the service is delaying the setup and execution of simulations unreasonably, and should be entitled to receive compensation.

Requests that complete successfully in under 30 seconds should be regarded as being successful. However, if the threshold for an intolerable delay is set at 30 seconds, then this is the longest that the client can be expected to wait before deciding that the request has failed and should be reattempted. Therefore, requests taking longer than 30 seconds to complete should always be regarded as failures. The failure mode is defined at line 354, pg. 299.

In order to configure and execute a simulation, a chemist will normally have to interact with the service around five times. With only five requests to make, and assuming some degree of forward planning on the part of the chemist as to the simulation they wish to execute, this whole process should not take more than a couple of minutes, assuming the correct operation of the service.

A failure may take up to 30 seconds to become evident, before it is treated as an overdue request, or it may become evident very rapidly. In the latter case, failures have the potential to waste almost as much time as overdue requests, hence a similar constraint of a maximum of 10 failures permitted within 10 minutes would be appropriate. In the former case, failures will occur more rapidly, so the chemist may be expected to persist to some degree in their attempts to access the service. However, again, a cap of 10 failures will tend to indicate a problem with the service. Therefore, I have concluded that the tolerance of failures should in this case be the same as the tolerance for seriously overdue requests (which must also be regarded to be failures), a maximum of 10 within a 10 minute window. Again, observing such a

pattern of failures should entitle the chemist to submit a bug report, therefore entitling them to receive penalties related to unavailability.

Since no formal definition of the behaviour of `Polymorph Search Webclient` is available, I rely on informal failure mode definitions to describe failure behaviour of the service. Since the service produces webpages, failure behaviour will be of two kinds: a failure to return an HTTP response code of 200, indicating success, and a failure to return a message body with the expected contents. The former can be described in a single failure-mode definition (line 287, pg. 297), the latter requires a failure-mode definition per operation, starting at line 389, pg. 299. Again the failure modes are violation and availability dependent to capture interactions with the throughput and availability conditions.

Conditions on failure behaviour and serious delays are therefore implemented using a permanent, fixed-window, fixed-occurrences, service-behaviour condition clause associated with failure mode descriptions for bad HTTP responses, operation-specific functional failures, and serious latency failures. A fixed penalty is levied, because the chemist should be expected to responsibly submit bug-reports in response to such behaviours, and therefore benefit from penalties associated with unavailability instead of unreliability. This condition is specified starting at line 1043, pg. 313.

Two separate conditions are needed to cover nuisance delays and serious delays. If the client receives penalty payments relating to serious delays then they may also have violated the terms of the condition related to nuisance delays resulting in the need to pay multiple penalties. This may be regarded as problematic. One possibility would be to define a relationship between the two latency failure modes, such that if a request were a member of the more serious mode it could not also be considered a member of the less serious mode. I have not implemented this solution; instead I observe that the penalty for serious delays could be slightly reduced to cover the possibility that nuisance delay may also be deemed to be occurring.

Availability condition

This condition-definition implements condition 2 from the list on pg. 193.

A delay of only ten minutes in starting a simulation expected to last 24 hours does not seem overly problematic. Therefore the penalty associated with such a delay should be quite light. If the service continues to be intolerably slow for a longer period, the inconvenience to the chemist will increase. However, the Chemist will not wish to have to continually submit requests over this period to establish that the service is slow. Neither will IS or CS wish to become liable for penalties related to faults that could have been rectified if they had been notified of them.

Instead, it is desirable that the chemist should complain to IS of serious problems, and receive compensation for the amount of time taken to fix the problem. Therefore it was decided that SLA 1 should include an availability clause providing the client with the justification to issue a bug report related to the use of the setup pages if persistent serious slowness or faultiness is experienced. Such bug-reports may eventually be matched by bug-fix reports issued by IS within the lifetime of the agreement, and the client can receive compensation based on the duration of unavailability so defined.

The availability clause will be associated with the service-behaviour restrictions covering intoler-

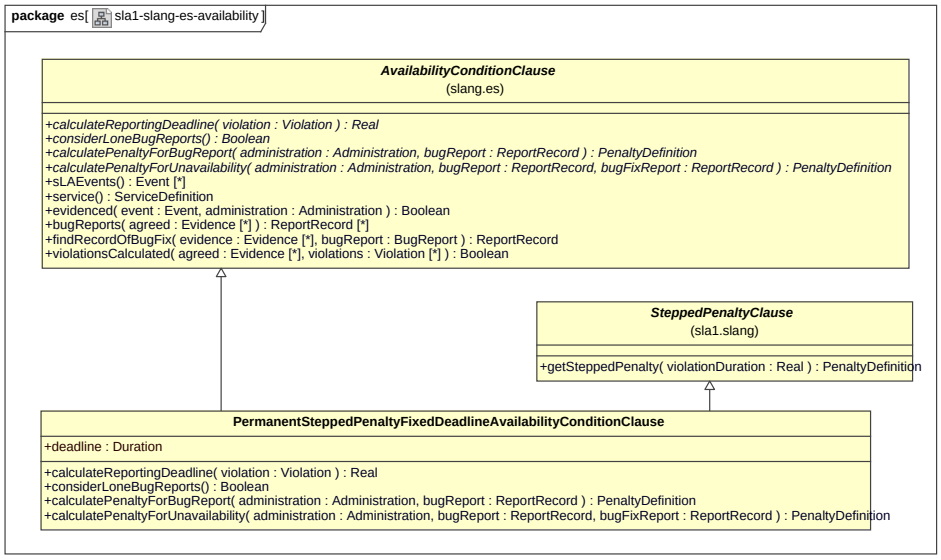


Figure 7.10: An availability clause type appropriate to SLA 1

able delays, and the production of erroneous results. The usage mode referenced by the clause will be the use of any operation. Since any operation may be critical to the configuration and execution of the simulation, if any operation is manifesting a persistent bug then the overall service should be considered to be unavailable.

An extension to `AvailabilityConditionClause` must be defined to determine the scheme by which penalties will be applied. The penalty for a period of unavailability should be related to its duration. I have therefore implemented a stepped-penalty availability clause.

I have decided that the client, Chemistry, will not wish to wait until a period of unavailability has come to an end before receiving compensation for it, so unterminated periods of unavailability are considered by the availability clause. The penalty for these periods should be related to the amount of time between the earliest of the start of the period of unavailability and the period of administration, and the end of the period of administration. The concept of an administrative period is not fundamental to administration-clauses. Therefore, I have introduced the concept of a consecutive administration clause, described below, that may be implemented by administration clauses which cover a set period of time.

I have specified a constant reporting deadline of 30 minutes for the availability clause, meaning that the client can legitimately report a bug relating to poor service, if they have noticed that the service has violated a reliability constraint within the last half-hour. The availability condition is defined starting at line 977, pg. 312.

Latency condition on completed simulations

This condition-definition discusses condition 6 from the list on pg. 193, and requires the definition of some sophisticated extensions to SLAng. I also consider the provision of guarantees concerning `MOLPAK` and `DMAREL` (condition 12).

The PS states that provided the chemist does not initiate more than a single simulation per day, then it should always be possible to complete the processing of a simulation within 24 hours. The

`Polymorph Search Webclient` does not notify the client when a simulation completes. Neither is the simulation-invocation operation a synchronous operation that takes 24 hours to return the result. However, since SLA 1 is intended to monitorable, the client will want to be able to: (i) gather some evidence to the effect that the results have not been prepared despite 24 hours having elapsed since the simulation started; and (ii) be able use this evidence to claim a penalty against the service provider.

The evidence potentially available to support the assertion that the results have not been produced are failures to execute the results retrieval operations successfully. The client must successfully access all eight distinct results pages for the results to be completely retrieved. Partial availability of the results is not adequate to establish that the simulation has been successfully completed. If, 24 hours having elapsed since a simulation was successfully started, the webclient experiences a period of unreliability, consisting only of requests for results that had not previously been retrieved, then these results may be considered unavailable and hence a penalty awarded.

In short, the webclient is implementing an asynchronous operation protocol. A latency failure occurs when the results are not available after the maximum permitted period for their production. The extensions I have defined in support of this are shown in Figure 7.11. An asynchronous failure-mode definition identifies a request operation, with a distinguished parameter identifying the batch of results to be produced. It also identifies a set of results operations, and their corresponding id parameters. For each operation, it identifies an associated success mode. Usages of the request operation in this mode trigger the production of results by the service. Usages of the results retrieval operations in their success modes indicate successful retrieval of results. The failure mode also identifies any number of reliability (behaviour-restriction) and availability conditions. Finally, the failure mode specifies a maximum permitted time in which results must be produced, and a deadline for their retrieval. The asynchronous latency failure mode then functions as follows: a usage of the request operation is in the failure mode if and only if it is a successful request for results to be produced, and a violation of any of the associated reliability or availability conditions occurs after the maximum time allowed for the production of results, but before the results have been successfully retrieved, and before the deadline for results retrieval has elapsed.

Clearly, what is required to guarantee the time taken to execute the simulation is a service-behaviour restriction condition related to an asynchronous failure-mode of this kind. However, there is another subtlety associated with this condition, which is that the Chemists themselves provide the programs `MOLPAK` and `DMAREL`. Since the overall time taken to complete the simulation is related to the performance of these programs, it is only safe for IS to insure such a latency if the Chemists make some reciprocal guarantee concerning the performance of `MOLPAK` and `DMAREL`.

Since this kind of scenario is likely to be fairly commonplace in grid-services, I have generalised the condition somewhat to consider failure-modes for delegated-execution services, in which the latency of the service depends on one or more executables provided by the client. Describing such failure-modes formally requires some extensions to the SLAng domain model, introducing the notions of executions, executables, processing nodes and slow-execution reports (a complaint made by the service provider to

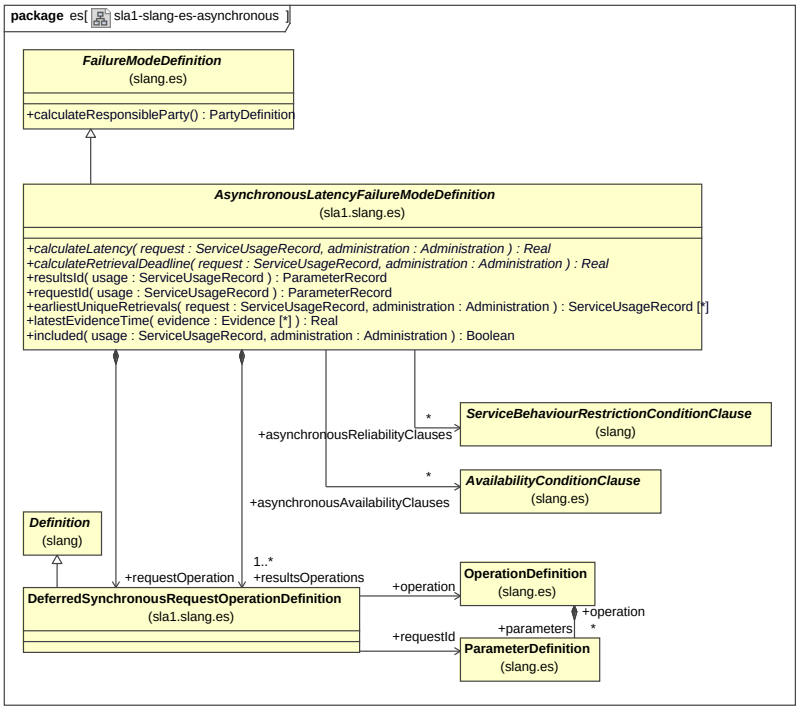


Figure 7.11: Clause-types for defining asynchronous electronic-service failure modes

the client that an execution took too long). These are shown in Figure 7.12.

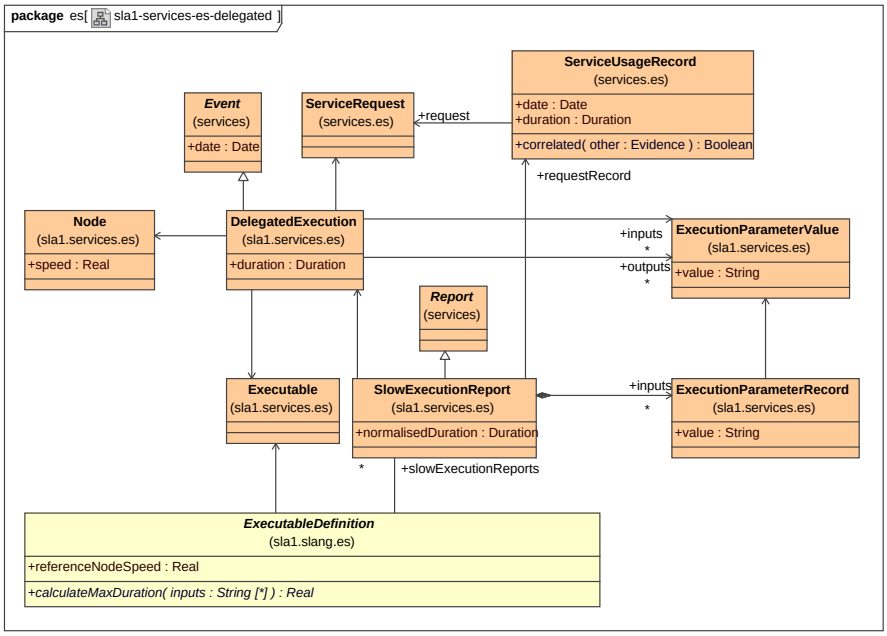


Figure 7.12: Domain-model extension describing the behaviour of delegated execution services

With reference to these semantic elements it is possible to formally define the meaning of the syntactic extensions shown in Figure 7.13.

An executable definition identifies an executable and provides a relationship between its inputs and the maximum time it will take to execute on a node with a particular reference speed (I have not

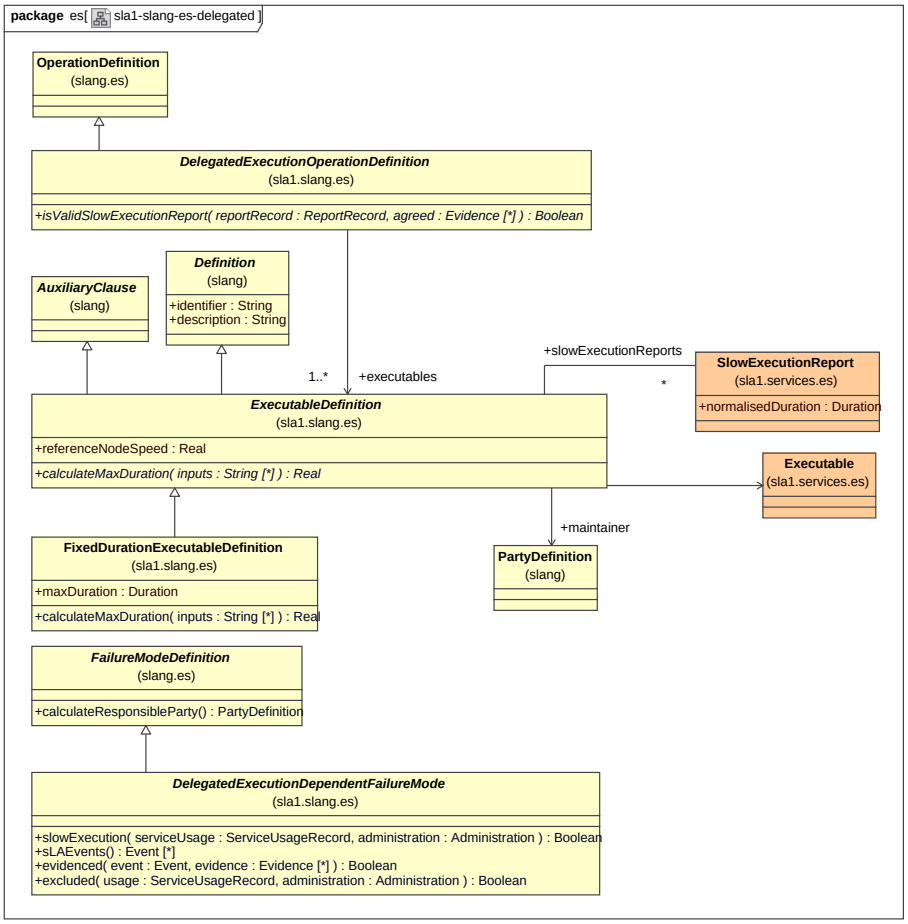


Figure 7.13: Clause-types for describing a delegated-execution electronic service

formalised the notion of speed other than as a simple scalar – more sophisticated extensions would need to consider different processing node architecture types, possibly including multi-processor architectures – this is not required in this case studies as the grid nodes are homogenously single-processor with uniform speed). The PS states that `MOLPAK` and `DMAREL` have fairly constant performance so I have provided a concrete extension of this clause type that allows the specification of a constant maximum time, independent of inputs.

Executables having been defined, a specialised operation-definition clause can identify executables as potentially being executed in the course of processing occurring as a result of a usage of the operation. Finally, a delegated-execution-dependent failure mode may be related to such operations. This abstract type of failure-mode does not indicate what outcomes should be regarded as failures, but does state that usages should not be regarded as failures in any more specialised mode if a slow-execution report has been submitted to the client in respect of the usage. Constraints associated with the delegated-execution operation definitions oblige the provider to only issue such reports when an execution caused by a usage has legitimately exceeded the defined maximum duration.

Having provided extensions for describing failures related to asynchronous operations and delegated-executions, it is finally necessary to combine these modes to create a latency failure-mode for

simulation execution. Note that simulation invocations shouldn't be regarded as being in this mode if the client has started too many simulations in a 24 hour period, or if the service is known to be unavailable, hence the failure mode will also be availability and violation dependent. The combination of failure-mode types ultimately required is shown in Figure 7.14. The simulation-failure mode is a fixed-latency (24 hours), fixed-deadline (one week to retrieve experiment results), delegated-execution (dependent on `MOLPAK` and `DMAREL` guarantees), availability-dependent (doesn't apply if the service is unavailable), violation-dependent (doesn't apply if the client has started too many experiments), asynchronous failure mode. This failure mode is defined starting at line 1102, pg. 315.

Failures of this type should never occur without implying a penalty for IS, so the mode is associated with a prohibited service-behaviour condition clauses, with a fixed penalty, at line 1102, pg. 315

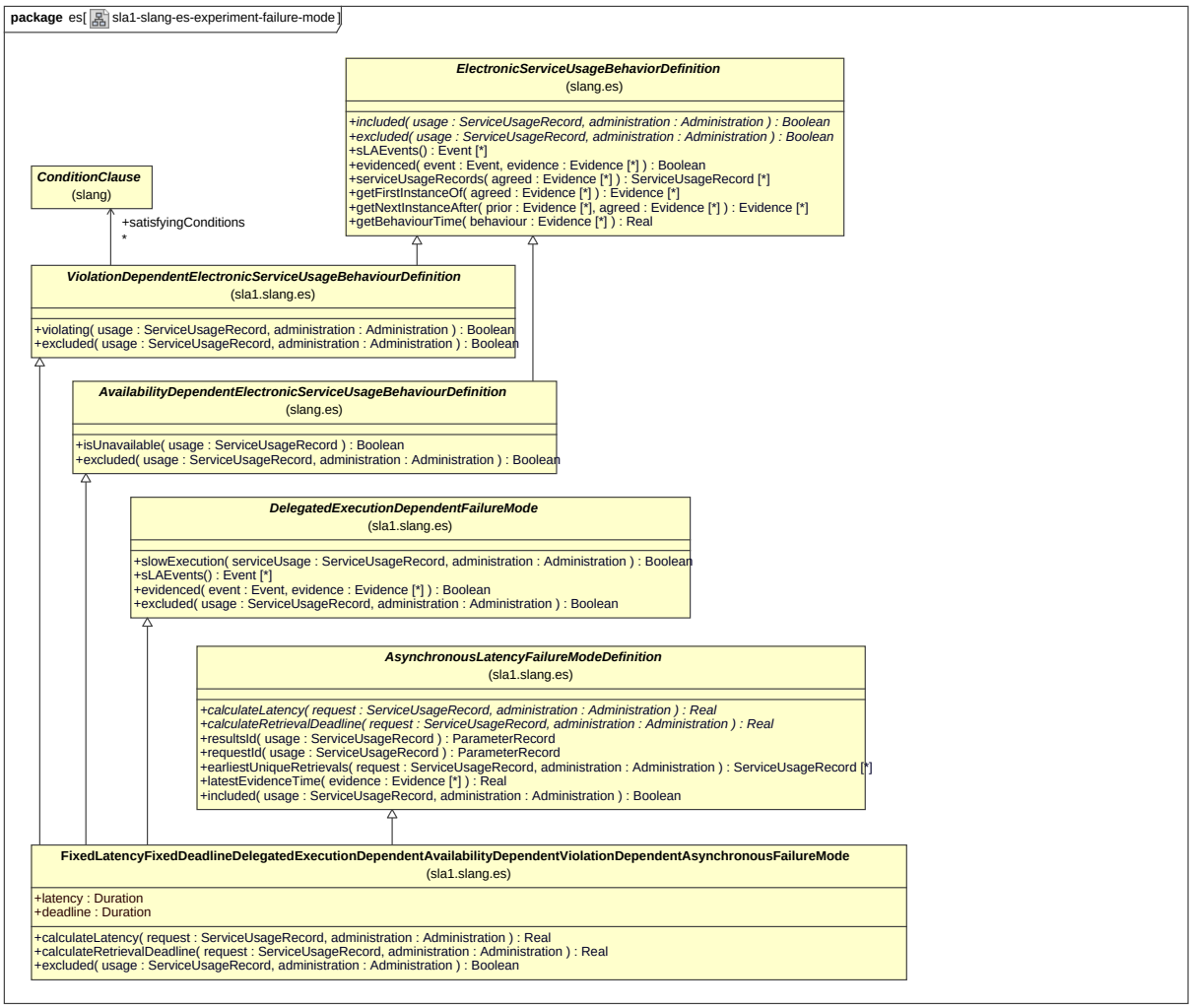


Figure 7.14: The 'simulation' failure mode, combining a number of more abstract failure-mode types

Administration and accuracy clauses

I have arbitrarily decided that the SLA should be administered once a week, on a Friday. I have implemented a lifetime for the agreement of one year. Since SLA 1 is mutually monitorable, reconciliation administrations, where the parties liaise to agree an account of service behaviour, are possible. Because

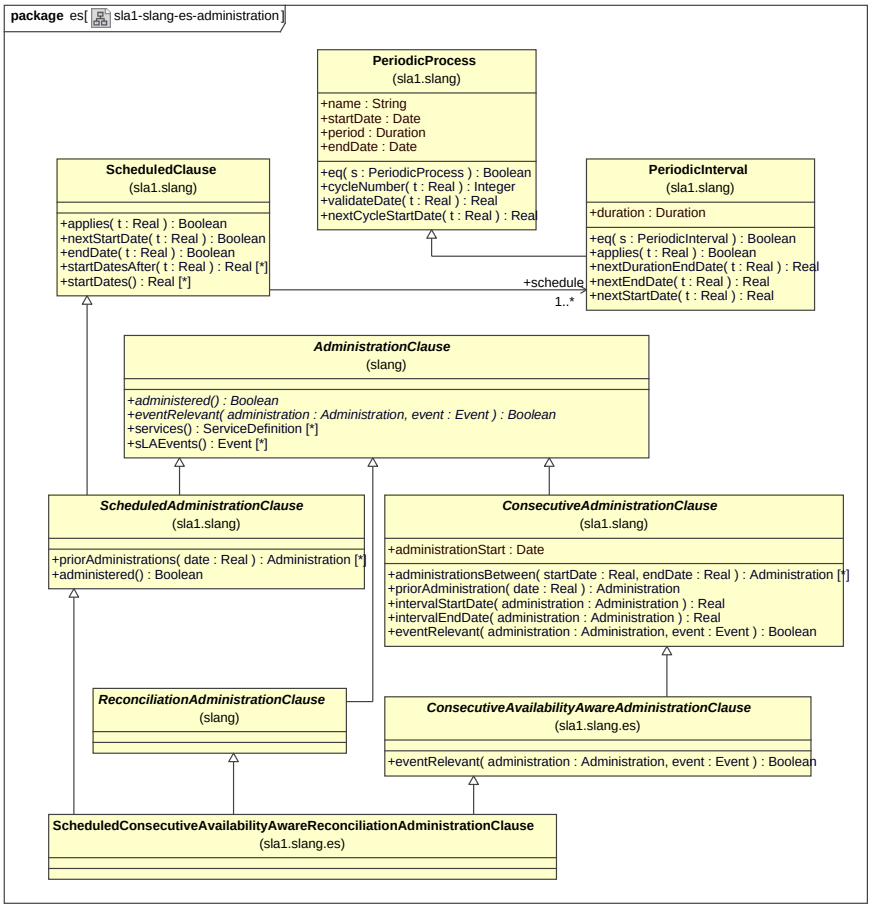


Figure 7.15: Abstract and concrete administration clause types for SLA 1

of the way that the availability condition calculates penalties, it is necessary that the administration be consecutive, in that it defines a sequence of consecutive administrative periods, the durations of which may be used in the calculation of penalties. In addition, the administration must be availability-aware, in that it should include evidence related to the exchange of any outstanding bug-reports (at the start of the administrative period).

Figure 7.15 shows the types used to extend SLAng's primitive notion of an administration clause to this effect. The routine administration clause is defined starting at line 929

Each administration covers all SLA conditions and must include evidence pertaining to all events relevant to the conditions occurring during the last administrative period, or since a specified start date for the agreement if there is no prior administration.

The inclusion of latency conditions in SLA 1 implies that the timing and duration of service-usages becomes a matter of concern. The SLA will therefore need to include a `ServiceUsageRecordAccuracyClause` to govern the measurement of this clause. Since the parameters of the latency conditions do not vary, the accuracy with which measurements are required will also not vary, and a `PermanentFixedServiceUsageRecordAccuracyClause` will be adequate. I have chosen a margin of error of 1 second, tolerably small in comparison the specified window of 1 minute for nuisance delays, and yet large enough to accommodate some clock-synchronisation

error. I have chosen 50ms as a suitable margin of error for the measurement of durations, as this is an order of magnitude greater than the resolution of most computer clocks (which should also experience negligible drift over the likely period of a service usage). I have stated that parties measuring these quantities should be 99% confident in their measurements, and the probability of a good log resembling a bad one should be 0.001%. In a log of 1000 measurements this permits only 9 errors, in comparison with a trusted log meeting the accuracy constraint, where an error is a difference greater than twice the error margin.

The timing of the exchange of various types of reports is also of relevance to the SLA, namely bug-reports, bug-fix reports, slow-execution reports, and as discussed below, termination reports. Therefore a permanent, fixed report-recording accuracy clause is also associated with the routine administrative clause.

Termination of SLA 1

Remaining to be considered of the conditions included in the list on pg. 193 are the termination conditions, 10 and 13.

It is possible that at some point during the one-year default term specified for SLA 1 that either Chemistry or IS will wish to withdraw from the agreement, signalled by the exchange of a termination report (a notion representing some kind of communication between the parties to this effect). As identified in the risk analysis for the SLA, the parties may wish to penalise this. Therefore I have implemented a fixed-penalty termination-by-report condition using the extensions shown in Figure 7.16.

The extensions include both a terminating condition, and a termination triggered reconciliation-adminstration clause, which must occur within a fixed deadline of the termination report being exchanged. The terminating condition, defined starting at line 1168, pg. 316, applies an equal penalty to whichever party chooses to pull out of the agreement early. The termination-triggered administration clause administers this condition, plus all of the conditions administered by the routine administration, for the period ending with the terminating administration. It is specified starting at line 1146, pg. 315.

The routine administration clause described above is sensitive to violations of terminating conditions, and administrations are not required by that clause after such a violation has been agreed in an administration.

7.7.2 SLA 4: Provision of the `plotws` web-service by the ISP to CS

The SLA clauses required in SLA 4 are:

1. a latency condition on plot operations;
2. a reliability condition on plot operations;
3. an availability condition on plot operations;
4. a payment scheme;
5. a termination penalty for the ISP cancelling the SLA;
6. an input-throughput constraint on operations;

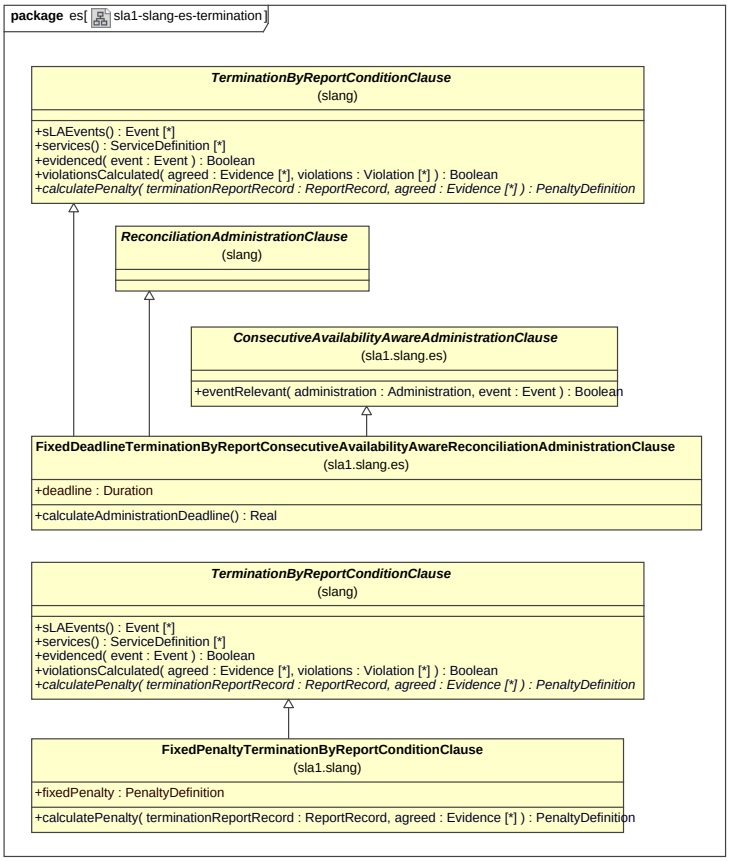


Figure 7.16: Administration and condition clause types related to the termination of an SLA, appropriate to SLA 1

7. a termination penalty for CS cancelling the SLA.

In addition it will be necessary for the parties to:

- agree a schedule of administrations;
- agree on standards of accuracy for gathering evidence related to service usages.

SLA 4 is considerably simpler than SLA 1, and I omit a detailed description of its definition here. The various plot operations for the service all function similarly to any of the operations of the `Polymorph Search Webclient`. Implemented in SOAP using an HTTP transport they can be described in a manner similar to a webpage request. Again I have relied on informal failure-mode descriptions. This is not problematic as the service should plot graphs according to well-understood representational principles which, it can be assumed, all parties are capable of understanding. Also, being mutually-monitorable, the parties will have a chance to negotiate what constitutes a violation during reconciliation.

Similar to SLA 1, SLA 4 contains a triumvirate of reliability (incorporating latency), throughput and availability clauses. The availability clause depends on the reliability clause to allow the client to issue bug reports. The reliability clause refers to failure modes that are dependent on the prevailing availability conditions and any violations of the throughput condition.

In the definition of extensions supporting SLA 4 I have duplicated several extension classes used in SLA 1. However, I have also taken the opportunity to demonstrate some variations. For the sake of the example, I have supposed that Southampton, and by implication also the ISP, would prefer an hour of maintenance time between midnight and 1 AM each morning, during which time they are not subject to either reliability or availability conditions. This I have implemented by associating schedules with failure modes, allowing failures to be excluded if the associate schedule does not apply. I have also implemented a scheduled availability clause, where time between bugs and bug-fixes only contributes to the calculation of the penalty if the schedule associated with the availability clause also applies (although bugs can still be reported at any time). These extensions are shown in Figures 7.17 and 7.18.

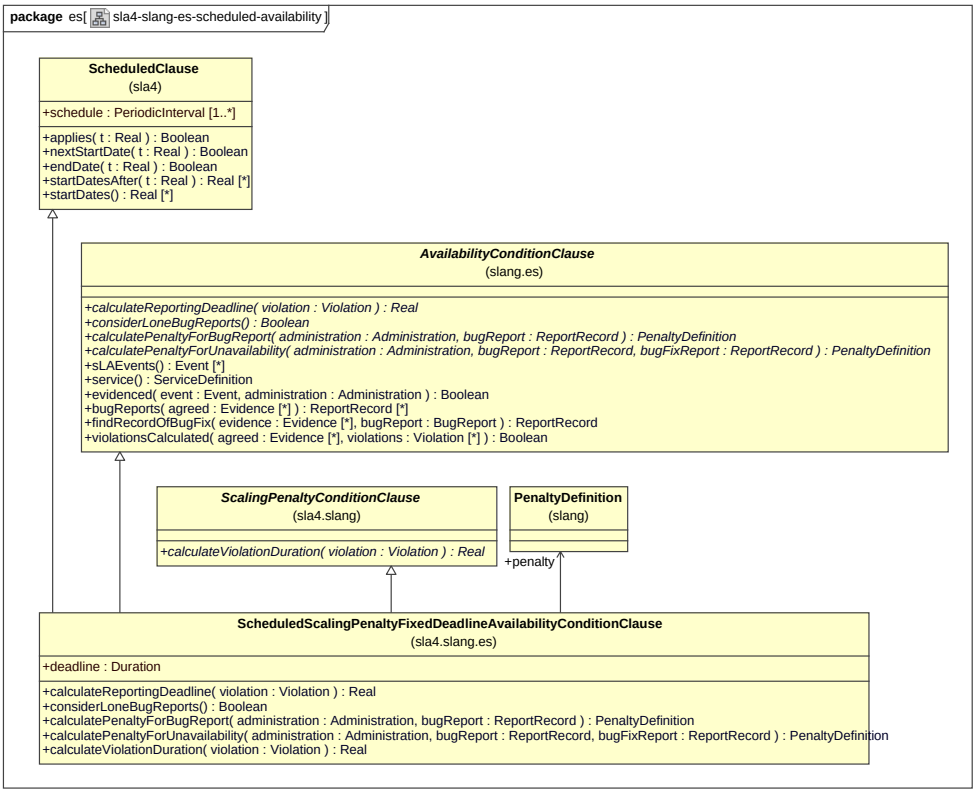


Figure 7.17: A scheduled availability type, guaranteeing availability only according to a specified schedule, in support of SLA 4

Finally, in contrast to SLA 1, I have assumed that Southampton prefers to calculate penalty payments related to violations occurring over an interval in a precise rather than stepped manner. Therefore, instead of using stepped condition-clauses as in SLA 1, I have introduced the notion of a scaling penalty definition. I have combined this with the notion of a penalty paid in Pounds Sterling.

Scaling penalties rely on the notion of a violation having a duration, which is not an essential feature of violations. Therefore to support the calculation of violation durations, it was also necessary to describe a category of condition clauses that define an notion of duration for their violations. A permanent, fixed-window, fixed occurrences, scaling-penalty, maximal service-behaviour restriction condition clause can then be used to define reliability conditions in the SLA. Extensions related to scaling penalties are shown

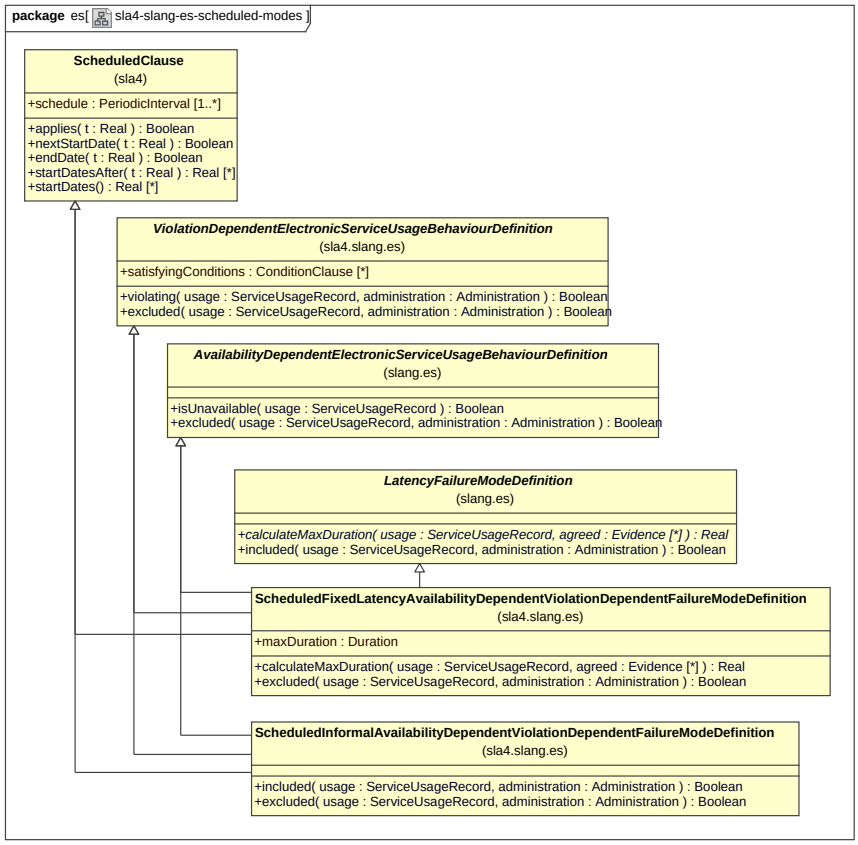


Figure 7.18: Scheduled latency and informal functional failure mode types in support of SLA 4

in Figure 7.19.

7.8 Case-study conclusions

The principle objective of this case-study was to demonstrate that SLAs appropriate to a realistic service-provisioning scenario could be defined using SLAng. In this respect, the case-study was successful. Two SLAs were specified using SLAng. In the next chapter, I argue that these SLAs are appropriate to the scenario as part of a broader evaluation of SLAng against the requirements that I established in Chapter 2.

Two notable observations arise from the case-study directly. First, that the base language provided good support for the definition of extensions (as will be described formally in the next chapter through an evaluation of the power, adequacy and specificity of SLAng) suggesting that the design of the core-language is based on valid assumptions and is of reasonable quality. Second, that although the case-study was not successful in producing a full set of prototype SLAs for the scenario, this was not due to deficiencies in the support developed for authoring SLAs, but limitations in the analysis and design of the SLAs. In fact, SLA 1 required very complicated extensions to SLAng, to support asynchronous operations, and conditions related to delegated execution. The capacity of the SLAng language core to be extended to express this SLA suggests both that the design of the language is appropriate, and that the approach of providing an abstract, extensible language for this semantic domain is a practical one, and is robust in the face of unanticipated requirements.

The need to define sophisticated language extensions for SLAng in the case-study suggests that a

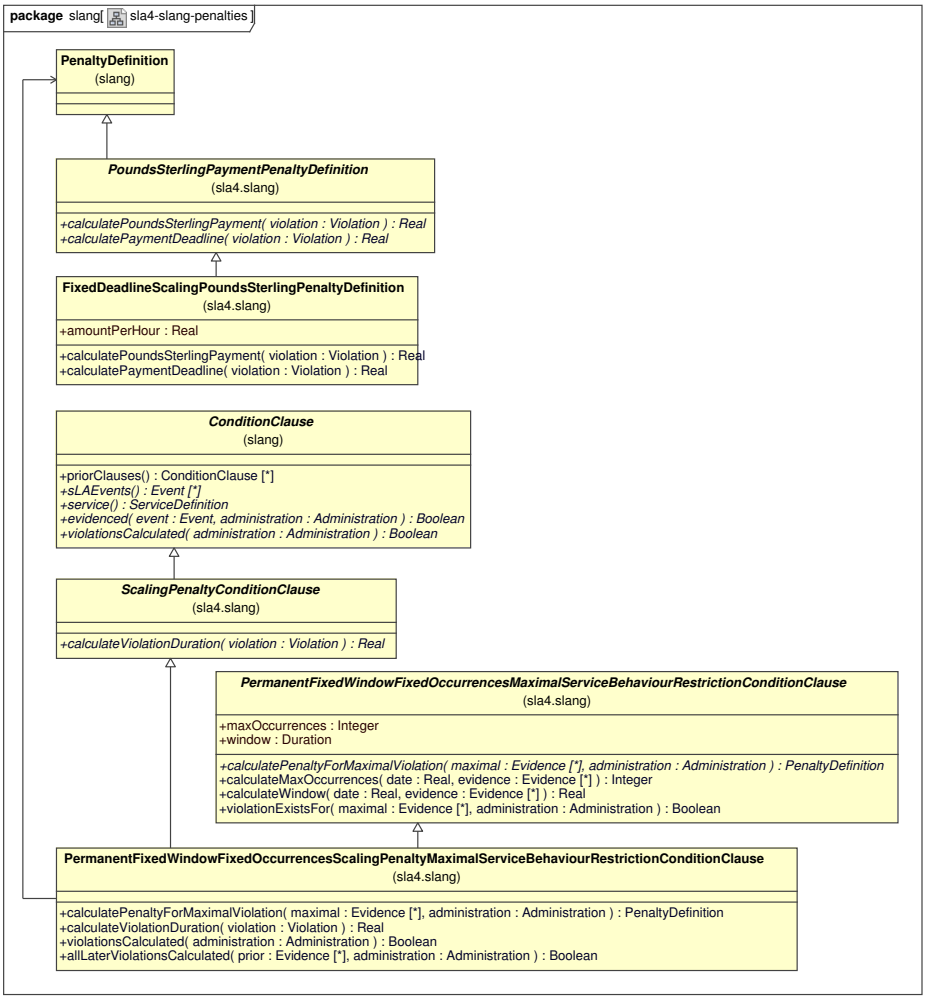


Figure 7.19: Condition clause and penalty definition types implementing scaling penalties for SLA 4

tentative generalisation can be made from the single ASP scenario examined: that SLAng should be appropriate for defining SLAs in different scenarios, provided that they are also ASP scenarios satisfying the criteria used to select the case scenario considered. Note that if SLAng were overly specialised to the case considered, then it would be reasonable to expect that extensions to the language would not be required. Therefore, the need for extensions suggests that SLAng is not specific to the scenario. Given this, the observation that SLAng was flexible enough to accommodate expressiveness requirements specific to the scenario, but not predefined, suggests that it will also be expressive enough to accommodate expressiveness requirements specific to other scenarios.

In order to generate additional insights from the case-study, it is also helpful to evaluate it as an effort to introduce new technology into the scenario. I consider two criteria: first, how useful the produced SLAs are to the scenario; second, how successful was the attempt to develop SLAs for the scenario using SLAng.

The result of the design stage of the case-study was a proposal for a system of SLAs containing 5 SLAs. As an approach to mitigating risks in the scenario, this system has both advantages and disadvantages.

The greatest practical disadvantage posed is the obligation imposed on the parties to monitor the SLAs at network boundaries. This poses the greatest challenge for the network service providers, IS, and the ISP, as they would have to implement new monitoring solutions at the edges of their networks (either by intercepting service requests or responses or providing proxy services). This monitoring requirement represents a hurdle to the adoption of this system of SLAs. However, as I demonstrated in Chapter 5, if highly monitorable SLAs are desired, and technology for trusted monitoring is not available, then there is no alternative.

Another minor problem with the system of SLAs proposed is that CS and IS enter into two SLAs (SLA 2 and SLA 3) where one SLA might be preferable. This may result in reciprocal payments resulting from the same violation, where a single SLA could have instead identified a violation without penalty. Similarly, it results in reciprocal guarantees being issued with respect to the behaviour of the executables `MOLPAK` and `DMAREL`. However, as discussed in Appendix B.2, some security guarantee would be required in any case by IS when allowing CS to specify executables to be run on cluster nodes during configuration.

In favour of the SLAs proposed is that having ruled out the use of SLAs to mitigate security risks, it was possible to propose SLA conditions for each of the SLAs such that all risks *intrinisic* to the scenario (not caused by the use of SLAs), were mitigated, and all *derived* risks (caused by the use of SLAs), were also mitigated. It was possible to do this without proposing changes to the scenario. This suggests that providing the parties were prepared to accept the monitoring and administration costs of using mutually-monitorable SLAs, then SLAs would be a practical risk-mitigation solution for this scenario.

The set of SLAs implemented suffers from several deficiencies related to a lack of knowledge of the case-study scenario. SLA 3 is not specified, because of the difficulty of determining the protocols by which the various `Condor` processes communicate. For the same reason, use-cases 2 and 3 were not elaborated, as they deal exclusively with this type of communication. Therefore, the SLAs proposed may not address risks related to the activities performed in these use-cases. Finally, the parameterisation of the concrete SLAs is suspect, because the true performance characteristics of the service are not known, and also economic significance of the service in terms of its operating costs and the magnitude of the financial risks implied by poor performance or termination of SLAs. Since SLAs 2 and 5 were shown to differ from SLA 1 and SLA 4 only in their participants and parameter values, I elected to produce only concrete examples of SLAs 1 and 4.

These deficiencies are all caused by limitations in the amount of effort that could be allocated to this case-study. It remains unknown whether, given more effort, these deficiencies could be rectified to produce a set of SLAs appropriate to the scenario, but it seems to me likely that they could. Certain aspects simply require more effort, such as the reverse-engineering of the `Condor` protocols (although it may be that if a strong desire to use SLAs in the scenario arose that it would be more convenient to restructure the service to avoid the need to do this, as discussed below). Other aspects may require the development of new theory. For example, a method to determine parameter values for SLAs must consider what performance analysis of a system is required, what financial analysis, how the results of

these analyses could inform the selection of parameter values, and how confidence in the properties of the resulting SLAs could be developed. These questions represent a challenge for future research efforts, but I think it would be unduly pessimistic to believe that progress on these questions could not be made.

It is the lack of methodological prescriptions for these aspects of the preparation of SLAs that is also the strongest criticism of the method that I followed to conduct the case-study. By regarding SLAs as a new technology being introduced to meet existing requirements, it was possible to develop a case-study method by analogy to software development, consisting of requirements-analysis, design and definition stages. The original assumption of this work, that SLAs are a mechanism for mitigating financial risks, suggested that requirements-analysis should focus on risk-analysis. Using use-cases to direct the risk-analysis, and subsequently designing conditions in relation to risks has apparently enabled a thorough approach to be taken, with the benefit of traceability, resulting in plausible SLAs which at least have the potential to mitigate those risks that have been successfully elicited. Assuming reasonable parameters could be determined for the conditions, further work would be required to test the SLAs in an operational context to determine whether they actually mitigated the true risks to the participants inherent in the scenario, or whether unexpected outcomes rendered the SLAs irrelevant.

7.9 Redesigning the service

Having proposed an initial set of SLAs for the eMaterials scenario, the final stage in the proposed case-study method is to consider what modifications could be made to the existing infrastructure to better accommodate the use of SLAs.

Clearly, a major hinderance to the production of a complete set of SLAs for the case-study has been the obscurity of the protocols by which `Condor` processes communicate. A `Condor` grid in aggregate potentially has at least two well-defined web-services interfaces, via which interaction is possible, one defined by `GridSAM`, and the other application-specific. However, in this case-study, the `Condor` implementation itself spanned an administrative boundary, with the `Condor controller` and `Condor submit daemon` controlled by CS, and the `Condor` nodes controlled by IS. Two solutions suggest themselves: either IS or some other support-function within the university should operate the grid completely themselves; or it will be necessary to clarify the means by which the various `Condor` process communicate, perhaps by re-implementing these communications using web-services. In the long-run, the former seems the more viable solution to me: high-performance computing is likely to become a commodity within a university that multiple academic departments will wish to use. Therefore it seems sensible to bring it under centralised administrative control.

This would drastically simplify the specification of SLA 3 in the scenario, which would similar to SLAs 1 and 2. CS would be reselling a grid-service to Chemistry, via IS, wrapped in the `Polymorph Search Webclient`, with the minor additional functionality of collating and summarising the results.

Chapter 8

Evaluation

This dissertation supports the following thesis: it is possible to provide practical language support for Service-Level Agreements (SLAs) for Application-Service Provision (ASP) that is better than that provided by previously proposed languages constructed for this purpose in the following respects: it provides greater assistance in expressing conditions that mitigate the risks inherent in ASP; and disputes related to agreements expressed in this manner may be more easily resolved in such a way as to respect the original intent of the parties.

In Chapter 2, I have first described the risks to which the parties in a typical ASP scenario are exposed, and hence requirements for systems of SLAs capable of mitigating those risks. I have then enumerated requirements for SLA languages and the specifications of such languages, which, if met, would result in a practical language, capable of expressing systems of SLAs containing conditions that would act in combination to mitigate these risks. Such a language would also permit the expression of SLAs that were both highly precise and monitorable, qualities that I have identified as contributing to the ease with which disputes could be resolved by observing that in order to resolve a dispute the intent of the original agreement must be retrieved from an SLA, and then used to pass judgement in relation to a set of evidence that has been obtained in a trustworthy manner.

In subsequent chapters I have introduced theoretical innovations in the design and specification of SLAs languages which made meeting these requirements possible. I then incorporated these innovations into the design of an abstract, extensible, domain-specific language for ASP SLAs, SLAng.

In the previous chapter, I have demonstrated that it was practical to use SLAng to specify SLAs in a realistic ASP scenario, using a case-study. In this chapter I conclude the demonstration of my thesis by arguing that the support provided by SLAng in specifying these SLAs, and by implication SLAs that might be specified in similar scenarios, is superior to that which would be provided by previously proposed languages designed for the same or similar purposes. I achieve this by first evaluating SLAng according to my requirements. I then survey alternative languages and argue that they do not meet my requirements to the same extent as SLAng does. Deficiencies identified in the languages commonly include a lack of support for expressing conditions, such as latency, reliability and throughput, that are clearly essential to mitigating the risks inherent in the ASP scenario, a lack of precision in the definition of their semantics, compounded by the lack of a clear definitive specification, and a disregard for considerations of monitorability, either in relation to the gathering of reliable evidence or the treatment of

measurement error when determining violations.

In addition, I further investigate the power, specificity and adequacy of SLAng, by evaluating the metrics defined in Section 4.5 in the context of the SLAs and language extensions developed during the case-study. This serves to demonstrate that SLAng is a highly powerful language and also extremely specific to its domain. However, its adequacy can be improved by the incorporation of extension elements found to be useful in common practice. I demonstrate that, by incorporating common elements from the SLAs developed in the case-study, SLAng's adequacy can be improved without compromising its specificity to a large extent, and use this result as a basis for a discussion of the future evolution of the language.

8.1 Evaluation of SLAng versus requirements

In this section I evaluate SLAng against the requirements stated in Section 2.8. I also evaluate language specifications, derived by extending SLAng, against the requirements for such specifications stated in Section 2.9.

8.1.1 Expressiveness requirements

I first consider SLAng's expressiveness requirement, stated as follows:

Language 1, pg. 44 – Expressiveness

The language must be capable of expressing all SLAs in a system of SLAs meeting the requirements specified in Section 2.7.

A full evaluation of SLAng with respect to this requirement involves considering the extent to which it is possible to express SLAs meeting the SLA requirements stated in Section 2.7. This I now do, with reference to the SLAs created as part of the case-study.

SLA 1, pg. 40 – Service conditions

The system of SLAs should entitle the client to either receive compensation, vary some SLA or SLAs in an agreed manner, or provide them with the opportunity to quit the system of SLAs without penalty, when the behaviour of the service, in so far as this effects the client, violates some anticipated requirement of the client, potentially including timeliness and reliability requirements.

The capability of SLAng to express systems of SLAs meeting this criteria was partially demonstrated in the case-study. SLA 1 was the only SLA in the system which addressed the client's need for compensation, and this SLA was specified in its entirety. The SLA includes timeliness and reliability conditions, and also an availability condition related to these conditions which serves to improve the practicality of the SLA and reduce its exploitability. These conditions are related to the definition of financial penalties that are applied to the parties in the event of condition violation. Conditions relating the exchange of a termination report to the termination of the agreements are included in both of the example SLAs.

SLAng required extensions to deliver this capability. However, the extensions required to express these conditions benefitted substantially from the contributions of the base-classes that they extended. In Section 8.1.3 I measure how adequate SLAng was to the case-study SLAs.

The expression of constraints relating to real-world behaviours was not demonstrated in the case-

study, because the case-study service exhibits no behaviour unrelated to its electronic services. However, the SLAng language core is designed to support the specification of such conditions. Base classes such as `ServiceBehaviour` and `ServiceBehaviourRestriction` are independent of any semantic elements relating specifically to electronic-services. Moreover, the case-study demonstrated the addition of new concepts to the domain model, such as `Executable`, that are not uniquely associated with electronic services. The combination of these two facilities in the language suggests the possibility to model real-world behaviour and specify conditions in relation to it.

SLA 2, pg. 40 – Client conditions

The system of SLAs should entitle any service providers involved to either receive compensation, vary some SLA or SLAs in an agreed manner, or provide them with the opportunity to quit the system of SLAs without penalty, when the behaviour of the client, in so far as it effects the service, violates some anticipated requirement of the provider, potentially including request-throughput limitations.

SLAs produced in the case-study demonstrated the capability of the language to express the requirements of service-providers, in addition to those of clients. In particular, throughput conditions were implemented, including an unusual condition in SLA 1 relating only to the throughput of successful requests. Although the specification of a complete set of SLAs for the case-study was prohibited by the difficulty of reverse-engineering the `Condor` communication protocols, the potential of SLAng to specify such a system is clear.

SLA 3, pg. 40 – Charging

The system of SLAs should make the service provider and network-service provider liable to receive compensation, in return for their contributions to providing the service to the client at the client's preferred point of service delivery, if the providers require compensation.

The potential of SLAng to implement charging schemes was demonstrated in the SLAs produced by the case-study, in which throughput constraints were used to associate financial penalties with service requests to implement per-use charging schemes. Clearly more complicated charging schemes would not necessarily be able to rely on this use of existing facilities in the language for specifying conditions. However, it would be straightforward to implement other schemes in language extensions, and the semantic definitions of these extensions could reuse existing concepts in the domain model, such as timed events, violations and compensation.

SLA 4, pg. 40 – Termination

The system of SLAs should make any party liable to receive compensation when one or more SLAs in which they participate are terminated prematurely by another party.

Both SLAs developed for the case-study included precisely defined termination conditions, and administration clauses that precisely define the obligations for the parties in determining final penalties in the event of early termination.

SLA 5, pg. 41 – Protectability

All SLAs in a system of SLAs must be protectable.

Protectability is difficult to assess in the abstract. Hopefully it will be possible to monitor the

use of SLAng in a real service-provision scenario in the future. Disagreements over SLAs occurring during such a study would provide the opportunity to assess the extent to which SLAng contributed to protecting the initial intent of the parties. However, I argue that because SLAng satisfies the precision and monitorability requirements, discussed below, to a high degree, then the SLAs expressed in SLAng will tend to be highly protectable.

SLA 6, pg. 41 – Understandability

SLAs must be understandable, so that all parties can verify that an SLA correctly captures their intent with respect to the agreement, and so the intended effect of the agreement can be easily retrieved in the event of a disagreement related to the award of penalties.

I discuss the intrinsic understandability of the SLAng language below. However, according to my recommendations in Chapter 4 for concrete statements expressed using a domain-specific language, individual SLAs expressed using XMI or HUTN should be understandable because they include a comment in natural language referencing the concrete-syntax standard in which they are written, as described in Section 3.2.3. It is therefore possible for a user to examine that standard, and subsequently interpret the SLA. This activity will involve retrieving and interpreting the language specification defining the abstract syntax and semantics of the language. This specification will be interpretable in a similar manner in relation to the meta-language specification in which it is defined, in the case of SLAng, a combination of EMOF, OCL and natural language.

These features of SLAng SLAs combine to eliminate any practical barrier to obtaining an interpretation of a SLAng SLA according to the SLAng language specification. SLAng SLAs are nevertheless technical artifacts that may require some expertise to interpret. SLAng is a powerful language, so a thorough understanding of a SLAng SLA will require more effort devoted to interpreting the SLAng language specification than the SLA document itself, which is principally a repository for parameter values. Some of these parameter values may be reasonably easy to interpret informally, such as the maximum latency value in a latency failure-mode description, or the window size and maximum occurrences in a behaviour-restriction condition clause. Such informal interpretation of an SLA is aided by the possibility of representing a SLAng SLA using the Human-Usable Textual Notation (HUTN).

SLA 7, pg. 41 – Precision

SLAs must be precise, so that their intended effect is unambiguous in the case of any disagreement related to the award of penalties.

The precision of SLAng SLAs depends on the precision of the SLAng language, discussed below, and on the precision of the concrete-syntax syntax used to encode the SLA. A SLAng SLA is expressed according to some concrete-syntax standard that permits the interpretation of the SLA document as a collection of objects conforming to the types in the SLAng abstract syntax. Existing concrete-syntax standards for this purpose, such as XMI and HUTN, are highly unambiguous, in the sense that a single document can only be reasonably interpreted as a single system of objects. The explicit referencing of both the concrete-syntax standard in which an SLA is written and the language specification to which it conforms, recommended in Section 3.2.3, not only provides a route to understanding an SLA, but also

eliminates any ambiguity that may be introduced due to the possibility of selecting an incorrect authority by which to interpret the document. Precision is also aided by the inclusion of a URI attribute in the `SLA` class, which allows the definitive form of an agreement to be unambiguously referenced.

SLA 8, pg. 41 – Monitorability

The system of SLAs should be as monitorable as possible.

SLAng SLAs may be mutually-monitorable, as in the case-study. Extensions to the core language may be defined to implement more monitorable SLAs (for example, arbitratable SLAs), and these extensions would benefit from the reuse of existing concepts in the SLAng domain model, such as timed events, evidence and administrations. It has yet to be shown that the monitoring requirements implied by SLAs that are more than mutually monitorable can be safely met in the ASP scenario. SLAng SLAs therefore represent the current state-of-the-art in monitorability.

SLA 9, pg. 42 – Error

SLAs should accommodate measurement error and uncertainty, either by only setting conditions on measured or agreed quantities, with a description being given of how the measurements are to be taken or the agreement reached, or by specifying acceptable degrees of confidence and margins for error on constraints over actual physical quantities.

The requirements for the calculation of violations related to conditions included in SLAng SLAs are defined in terms of evidence used by the parties during administrations of an SLA. Conditions are defined such that the association of a condition requiring a particular type of evidence with an administration clause implies that an accuracy clause pertaining to that type of evidence must also be associated with the administration clause. Accuracy clauses require that all evidence used during administration meets a specified standard of accuracy, according to the accuracy constraint developed in Section 5.2.1. This constraint is approximately monitorable by both parties to a mutually-monitorable SLA.

SLA 10, pg. 42 – Feasibility

SLAs should only include conditions for which violations can feasibly be calculated, given all pertinent evidence.

Because SLAng currently relies on extensions to express most SLAs, it is not possible to guarantee that this requirement is always met. However, work discussed in Section 4.4.3 includes two successful efforts to generate violation-calculating monitoring systems based on fixed sets of SLAng conditions, including a method based on timed-automata that can calculate violations of reliability and throughput conditions in linear time. As discussed in Section 4.4.2 it would be desirable to use the SLAng language specification directly as part of a system to calculate violations based on evidence of service behaviour. Early efforts to achieve this have not been successful as the semantics of SLAng is currently formalised in a manner that is not amenable to efficient interpretation over data-sets of a realistic size. However, since there is clearly no theoretical impediment to the efficient monitoring of common conditions such as reliability or throughput, and OCL offers some flexibility in formulating invariants and side-effect-free operations, I believe that future work in reformulating the SLAng semantics has the potential to address this issue.

SLA 11, pg. 42 – Cost

SLAs should be as cheap to produce, protect and administer as possible.

SLAng SLAs are both highly formal, and typically require the definition of extensions to the core language. These characteristics tend to increase the cost of production of SLAng SLAs, as demonstrated by the case-study which required an extensive analysis effort to inform the production of the SLAs, and the definition of language extensions of a similar size to the SLAng core language itself. Naturally, these activities also require expertise to complete, which may not be commonly available.

However, the characteristics that increase the cost of preparation of SLAng SLAs were all introduced to meet other requirements that seem to be essential to the production of quality SLAs, such as the ability to express the conditions that are actually required to mitigate the risks that parties experience in an ASP scenario, and to do so in a precise manner. It therefore seems reasonable to consider the cost of preparation of a SLAng SLA as being analogous to the payment of the premium on an insurance policy. If greater protection is required, a higher premium must be paid.

The cost of preparation of SLAng SLAs is diminished to some extent by the restrictiveness of the language, and by its automatability, which allows the automatic generation of an editor component, and also consistency checking of SLAng SLAs.

It is undesirable for SLAng SLAs be considered to be only an option for high-value service relationships. Instead, the core language should be augmented with vocabulary from commonly-required extensions in an effort to increase its adequacy without drastically decreasing its specificity and therefore resulting in a bloated and unusable language. I demonstrate the potential that SLAng offers as a starting point for this type of evolution in Section 8.4.

SLA 12, pg. 43 – Machine readability

SLAs should be expressible using an intrinsically machine-readable syntax. This requirement should not compromise understandability.

SLAng SLAs are expressed in an intrinsically machine-readable manner, as demonstrated by the development of the UCL UML tools to assist in the specification of the SLAng language and the automated generation of tool-support capable of editing SLAs and checking them for consistency. As discussed above, the possibility of expressing SLAng SLAs using the HUTN maintains a tolerable level of understandability for SLAng SLAs.

SLA 13, pg. 43 – One definitive form of agreement

If multiple forms of an SLA exist, they should be provably equivalent, or it should be clear which is the definitive form.

The SLAng `SLA` class includes a URI attribute that should be used to refer to a location at which the definitive form of an SLA should be accessible to authorised parties.

SLA 14, pg. 43 – Non-exploitability

SLAs should be not be exploitable.

It is not clear how exploitable SLAng SLAs are. This is an important topic for future theoretical and practical investigation. However, SLAng SLAs do have features that potentially contribute to reduc-

ing exploitability. SLAng includes support for expressing throughput constraints, which allow service providers to restrict the extent to which a client can exploit the limited capacity of an electronic service in order to obtain penalty payments. It also allows the expression of availability constraints that can be used to promote an exchange of information between the parties regarding any faults, thereby reducing the possibility for either party to exploit information of this kind.

Entering into an SLA can be regarded as entering into a game in which the parties compete to obtain the greatest entitlement to penalty payments, while trying to avoid incurring costs of various kinds (for example, associated with lost business opportunities). In the future it will no doubt be useful to examine how different SLAs can affect the strategies that may be applied by the parties in the scenario, and hence determine whether any party can gain an unfair advantage. Such a theory would also have to be supported by empirical studies to determine whether the theoretical model correctly reflects the tactics that it is possible for a party to apply. Such investigations may be aided by the benefits provided by SLAng with respect to analysability, discussed below.

SLA 15, pg. 43 – Analysability

SLAs should be amenable to analysis to reveal implications that are not explicitly stated.

I have not discussed analysability extensively in this dissertation. However, in previous work I described the potential for SLAng SLAs to be used as artifacts in performance analysis activities [119]. This work is in its infancy. However, the advantages that SLAng provides in terms of analysability can be considered to delivered by two main features of the language: SLAng SLAs can be automated in various activities related to analysis, including testing and consistency checking (albeit with some deficiencies related to feasibility, as discussed above); and SLAng benefits from a model-denotational semantic definition.

The automatability of SLAng, discussed further below, is potentially of use in testing SLAng SLAs for particular properties. For example, if the conformance of a particular set of events to a SLAng SLA is in doubt, then it may be checked. Such tests can be used to generate insight into the implications of a SLAng SLA. If the feasibility of interpreting the OCL components of the SLAng specification can be improved, then this type of analysis may be performed on a larger scale, for example automatically administering an SLA as a component in the simulation of a service scenario, to check risk mitigation or exploitability properties.

In pre-existing, alternative work on language for SLAs, the importance of SLA information in performance analysis activities has been emphasised [28, 69]. In [119] I distinguish between *inter-service* composition and *intra-service* composition. Inter-service composition may be supported by analyses that match requirements to SLA conditions. If the requirements are expressed as desired conditions, then SLAng naturally defines a notion that I call *SLA compatibility*. An SLA, A, is compatible with an SLA B if all behaviours implying a violation of B also imply a violation of A. This implies that all behaviours acceptable to A will also be acceptable to B. Compatibility provides a very strong standard for matching SLAs, but is hard to reason about due to the expressive power of OCL.

Intra-service composition involves determining the overall QoS characteristics of a service based

on its components, some of which may be services with SLAs attached. This is an extension of standard performance analysis for services. Previous work on using SLA information to assist in this problem, for example [28], occasionally overlooks the fact that SLAs do not guarantee performance properties, but rather that either performance targets will be achieved or a party will become entitled to compensation. However, SLA information can be useful to performance analysis if assumptions concerning the likelihood of parties meeting the expressed conditions are added.

The inclusion of a domain model in SLAng duplicates, and was partly inspired by, the practice of providing a domain model in OMG specifications that are intended to support the analysis of models, in particular the UML Profile for Schedulability, Performance and Time Specification [89]. The primary advantage of such models is in the precision they lend to the semantic specification of a language. A clear understanding of these semantics in clearly essential when implementing tools that perform analysis on artifacts of the language. However, I believe that this approach partially anticipates a future requirement for languages in model-driven developments. When multiple domain-specific languages are used, the requirement to integrate information expressed in diverse languages into analysis processes will inevitably arise, and consideration must be given to how this integration can be assisted automatically. This is the case when attempting to reason about intra-service service composition in an MDA development. Naturally, many theoretical challenges relate to integrating information from different sources and reasoning about the validity of inferences derived from the combined information. However, explicit, machine-readable models of the semantics of the source languages in which the information is expressed will clearly be assets when providing automated assistance in such tasks.

8.1.2 Remaining requirements for ASP SLA languages

Having considered how SLAng meets the expressiveness requirement for an ASP SLA language in the previous subsection, I now consider how it meets the remaining requirements that I identified for such languages:

Language 2, pg. 44 – Understandability

To understand an SLA written in an SLA language it is necessary to understand the language. The language should be structured so that it is easy to understand.

The understandability of the SLAng language is party dependent on the understandability of the SLAng language specification, discussed below.

Without attempting an empirical study, and from the subjective viewpoint of the designer of the language, it is difficult to assess how understandable SLAng is. However, I believe that the understandability of the language is enhanced by the following features, which are related to the language itself rather than how it is specified:

The semantics of SLAng are defined at the level of abstraction of services, events, parties and evidence, rather than relying on a more abstract mathematical concepts to describe the concepts to which the language relates. This should make it easier for people familiar with the domain of electronic services to understand what is intended by a SLAng SLA.

SLAng is also highly specific to specifying ASP SLAs, as discussed in Section 8.1.3. Hence, a user

attempting to interpret a SLAng SLA will not be distracted by irrelevant features of the language.

Language 3, pg. 44 – Precision

The meaning of an SLA is dependent on the semantics of the language in which it is expressed. Therefore, if the SLA is to be precise in its meaning, then the semantics of the language must also be precisely defined.

The meta-modelling approach used to define SLAng provides precision for the language in three ways: first, the syntactic structure of the language is precisely defined using an abstract-syntax model; second, the addition of a domain-model and the use of the model-denotation approach to define semantics for the language make it clear how SLAs relate to the real world – moreover this joint model can be automated to gain insight into how an SLA applies to a particular situation; finally, the close and systematic coupling of natural language descriptions of all elements, syntactic and semantic, in the language specification definitively establishes the correspondence between formal elements in the specification and real-world entities, and make it harder for a human user to misinterpret the formal elements.

Language 4, pg. 44 – Restrictiveness

The language should exclude SLAs that do not meet the requirements specified in Section 2.7.

Constraints included in the syntactic model of SLAng act to rule out illogical SLAs where this can be anticipated. Constraints include multiplicity constraints, such as that specifying that behaviour-restriction conditions must be associated with at least one behaviour, and OCL invariants applying within the syntactic model, such as the constraint that the reliability clauses referenced by an availability clause, establishing the conditions under which a client may issue a bug-report, must refer to failure modes occurring within the usage mode covered by the availability condition (so that unavailability in some usage mode cannot be established by the unreliability of requests in a disjoint usage mode).

The syntactic model also contains constraints designed to require good quality SLAs. For example, associating any type of electronic-service behaviour-restriction condition clause with an administration clause implies that a permanent, fixed, service-usage-record recording-accuracy clause must also be associated with the administration clause, to establish a basic standard of accuracy for recording service usages. Note that this constraint could have been omitted – a poor SLA might still be useful without a standard for accuracy. Alternatively, the SLA author could have been required to ensure that the requirement that all service-usages have a constrained accuracy was met without prescribing the use of a particular type of clause (note that other clauses can be used in addition to a permanent, fixed clause, to tighten the accuracy requirements under specific circumstances). This would have been more flexible, but provided a large opportunity to introduce flaws into an SLA.

All constraints in the SLAng specification can be checked, either for an SLA, instances of types of the domain model, or some combination of both, using a repository generated from the language specification using the UCL UML tools.

Language 5, pg. 44 – Ease of use

In addition to being easy to understand, the syntax should be easy to write, possibly with the aid of tools.

Currently support exists for authoring SLAng SLAs using the HUTN, which is intended to be easy for humans to use, or using a tree-structured JMI-repository editor for Eclipse allowing the specification of objects conforming to the types in the abstract-syntax of SLAng. It is probably fair to say that neither of these two approaches represent the apogee of usability for producing SLAs. However, they do demonstrate features, such as restrictiveness, and amenability to consistency checking, that contribute to the usability of SLAng. The repository editor also demonstrates the retrieval of documentation from the SLAng language specification, which is presented in tool-tips to provide context-sensitive help for the author of an SLA.

Language 6, pg. 44 – Power

Because the SLA language is only defined once, but may be reused in multiple SLAs, as much of the burden of expressing the SLA as possible should be placed on the SLA language, except where this is incompatible with requirements for understandability for either the SLA or the language.

As described in Section 8.1.3, SLAng, when augmented with SLA-specific extensions, is extremely powerful. For example, a power measurement of 0.97 for SLA 1 can be interpreted as stating that (at least) 97% of the information burden of the SLA is conveyed by the language specification rather than the SLA itself.

This result suggests that SLAs expressed in SLAng are quite concise.

However, such an observation cannot be used as a measure for the overall usefulness of the core SLAng language, because SLAng only obtained an adequacy measurement of 0.58 for SLA 1, which could be taken to mean that only 58% of the 97% (56% overall) of the information carried by the SLA was contributed by the language, although it is not at all clear that the information provided by the language has the same value as that provided by the extensions.

In Section 8.4 I discuss improving the adequacy of SLAng by incorporating extensions that are not essential to all ASP SLAs, therefore potentially compromising the specificity of the language.

Language 7, pg. 45 – Automatability

It should be possible to produce tools that take SLAs expressed in the language as their input. The tools should rely for their functionality only on the specification of the language, so that anybody who has access to the language definition can reuse the tools successfully.

SLAng is automatable to a high degree. It is possible to generate a repository for authoring SLAng SLAs, checking the consistency of SLAng SLAs, and testing the conformance of scenarios to SLAng SLAs. However, due to problems related to the feasibility of checking large scenarios described above, the automatability of SLAng could be improved.

Language 8, pg. 45 – Analysability

The semantics of the language should be oriented towards that of known analysis models, provided this is compatible with expressing the true requirements of the client, and any additional constraints required to avoid exploitability.

SLAng does not really meet this requirement to a high degree. As discussed in Section 6.1, I rejected the notion of basing the semantics of SLAng on a process algebra as I felt that the simplifying

assumptions on which such algebras depend are incompatible with specification of a precise agreement concerning the way in which real services should be delivered. However, as discussed above, SLAng to some extent anticipates future analysis techniques that will be needed in model-driven development activities, by providing an explicit domain model.

The SLAng language as it is manifested in the SLAng language specification is also amenable to analysis, as a result of being defined using an object-oriented meta-modelling language, which can be regarded as a common theory of objects. The metrics used to measure the language in Section 8.1.3 are an example of such an analysis technique.

8.1.3 Requirements for ASP SLA language specifications

I now consider how language specifications, derived by compiling the SLAng specification sources with extensions, and compiling the resulting language into an XMI document, per the recommendations described in Section 6.11, meet the specification requirements described in Section 2.9:

Specification 1, pg. 45 – Completeness

The specification should fully define an SLA language meeting all of the requirements specified in 2.8.

As discussed in Section 6.11, SLAng, as an abstract language, requires a representation that can easily be used as the basis for extensions. At present that is best provided by the custom syntax provided by the UCL MDA tools for language specifications based on EMOF and OCL, as this syntax allows the modularisation of a specification into several files. It would however be preferable if SLAng could be definitively represented using a standard concrete syntax, such as XMI.

Since SLAng is an abstract language, its specification will never meet all of the requirements for an SLA language, due to the limitations on its expressiveness. The SLAng language is fully defined by the SLAng language specification. However, the SLAng language itself does not fully meet its expressiveness requirement, in that it must be extended to specify most SLAs.

Because SLAs expressed in SLAng must refer to a single language specification, every time extensions are included with SLAng, this effectively creates a new language. As a consequence of type and syntax checking the sources for this language, an XMI file can be produced, and this should be regarded as the specification for the extended language. Such a specification will necessarily be complete for the statements that will be expressed using the extended language.

Specification 2, pg. 45 – Understandability

The specification must define the SLA language in a way that is understandable.

The SLAng language specification is the combined EMOF, OCL and English description of the language provided in the non-standard syntax required by the UCL MDA tools, which resembles HUTN to some extent, but also a block-structured programming language such as Java. This syntax, although non-standard, is, I believe, reasonable easy for the average programmer to read or write. It may also be compiled to XMI, and this XMI representation is a suitable stating point for transformations to easier-to-read representations. The UCL MDA tools provide two tools for generating documentation from a language specification, in either HTML or LaTeX format. The LaTeX documentation tool was used to

generate Appendix E. Naturally, the same arguments apply to extensions of the language.

Specification 3, pg. 45 – Precision

The specification must define the SLA language in a way that is precise.

The SLAng language specification defines SLAng according to the simple but unambiguous object-orient type theory provided by the EMOF model, and also the OCL 2 specification. Its precision is therefore controlled by the quality of these standards to some extent.

The specification also relies of statements in English to definitively establish the meaning of elements in the domain model. The precision of these statements is difficult to assess. However, because the statements are defining simple correspondences between domain-model classes and familiar elements of an electronic-service provisioning scenario, it is to be expected that they are reasonably precise. This is one of the main intended benefits of adopting a model-denotational approach to defining SLAng.

Specification 4, pg. 45 – Automatability

The meta-language employed in the specification should be defined in such a way to assist the development of tools that rely on the SLA language definition, for example, by offering a formal definition of the SLA language that could be used as the input to software engineering tools.A

As previously discussed, due to its reliance of EMOF and OCL, the SLAng language specification, or specifications of languages extending SLAng, compiled into XMI, are highly automatable, for example as the input to a tool capable of generating JMI repositories.

8.1.4 Summary of conformance to requirements

It is perhaps unsurprising that I should judge that SLAng meets my requirements for languages and specifications to a high degree. I originally identified these requirements as being important then focussed my research and the design of SLAng upon meeting them. The usefulness of this subjective evaluation depends first on whether the arguments I provided in Chapter 2 to justify the inclusion of each requirement are valid, and hence result in a set of requirements that a genuinely useful SLA language must meet, and then on the validity of my assessment of the degree to which SLAng meets these requirements. I have attempted to support my conclusions by providing argumentation in support of both steps, both here, in Chapter 2, and throughout this dissertation wherever I have introduced theoretical innovations.

Nevertheless, I have identified certain areas in which the language is not as successful as I hoped. One is in the adequacy of the language, which can be improved as discussed in Section 8.4, suggesting that the language should be viewed not as a final and static accomplishment but as a starting point for the design of a more pragmatic language. Another is in the feasibility of checking SLA conditions using the language specification directly. I believe this can be improved by reformulating constraints in the language, but it may be that a more suitable constraint language will need to be investigated as a basis for the definition of the language. Finally, SLAng does not offer strong support for analysis of SLAs, except is so far as it has a formally defined semantics, and a specification that is amenable to measurement.

The ultimate evaluation of the language itself will be the extent to which evidence emerges as to its usefulness in the future. This may be indicated by adoption of the language as a basis for defining SLAs

in real ASP scenarios, or by the reuse of theoretical innovations described here in future improvements to the state-of-the-art for SLA languages.

However, the theoretical concern of this dissertation is not to demonstrate the quality of SLAng as an absolute, but rather in comparison to previous work, which is discussed in the next section.

8.2 Survey of related languages

Appendix A provides a survey of languages either intended for, or conceivably of use in specifying SLAs for ASP. Where I deemed sufficient information concerning a language to be available, I have attempted an assessment of the language against my requirements, which I have condensed into three main questions:

1. **To what extent does the SLA language provide support for expressing conditions to mitigate the risks involved in the ASP scenario?**

   This requires the ability to express reliability, latency and throughput constraints, and constraints on the real-world behaviour of the service, associated either with financial penalties or the right to terminate the SLA. Also to enable the provider to charge for the service, and to associate penalties with the decision by either party to prematurely terminate the agreement. The language should also not be exploitable.

2. **How do SLAs expressed using the language contribute to increasing the likelihood that a dispute concerning an SLA will be resolved according to the original agreement?**

   This includes being: understandable; precise; monitorable; having a principled approach to the treatment of measurement error; expressing only conditions, conformance to which can feasibly be calculated; and identifying the definitive form of any agreement.

3. **How does the design of the language and its specification contribute to reducing the costs of outsourcing activities, including the authoring of SLAs?**

   Is the language restrictive, powerful, easy to use, automatable and analysable?

I now summarise the main findings of this exercise.

The perception of a need to either describe or constrain the QoS properties of web services, CORBA services, or in general client/server services consistent with the model proposed in Section 2.1, has motivated a large amount of previous research. Recent work has ranged between that focussing on describing requirements for the quality-of-service for web-services, and that describing contractual obligations more generally. Older work focussed on describing QoS for CORBA systems.

Most recent approaches to defining SLA languages have provided, or asserted the availability of, an XML schema for their language. This includes: the Web-Service Level Agreement language (WSLA, Section A.1, pg. 249); the Web-Services Offering Language (WSOL, Section A.2, pg. 251); the Web-Services Management Language (WSML, Section A.3, pg. 252); the Rule-Based Service-Level Agreement Language (RBSLA, Section A.4, pg. 254); the Web-Services Agreement Specification (WS-Agreement, Section A.6, pg. 257); and the Business-Contract Language (BCL, Section A.7, pg. 258).

The use of XML is clearly intended to ease integration with other web-services technology, such as WSDL or SOAP that are also dependent on XML, and conveys some benefits related to automatability. Of these languages, only BCL advertises a non-standard human-usable notation as an alternative to XML, increasing its usability.

WSLA, WSOL, WSML, RBSLA and WS-Agreement all rely on the provision of extensions of some kind to permit the complete expression of an SLA, with WSLA, WSML and WS-Agreement providing abstract data-types in their schemas to guide extensions, and RBSLA and WSOL relying on the use of externally provided ontologies (although the precise requirements for these remain unclear in both cases). The languages surveyed provide very little support for expressing latency, reliability or throughput conditions. In each case, either syntax for such conditions is missing, and the expression of such conditions therefore relies entirely upon language extensions, or syntactic elements exist but are not accompanied by semantic definitions of sufficient precision to support the calculation of violations. The support provided by the abstract schema types is very scanty and nowhere is documentation provided offering any guidance in producing extensions. In contrast, SLAng provides base-classes that encode much of the required semantics for these types of conditions, with extensions required only to provide those details that are SLA specific. These extensions are largely straightforward to define as they involve the overriding of well-documented abstract operations. In addition, examples provided of the use of the languages are universally hypothetical, in contrast to SLAng, the expressiveness of which has been demonstrated in a case-study involving a real service.

All of the alternative languages suffer from imprecision due to a number of factors. Universally, a separation exists between the XML schema definition of the language, and the language specification document, or documents. This hinders traceability between SLAs and the definition of their semantics. WSLA, WSOL, WSML and WS-Agreement all have informally defined semantics expressed solely using natural language. The semantics for RBSLA are incompletely specified. WSOL, RBSLA and BCL have no definitive language specification document, and instead are described in collections of academic publications. Neither BCL, not EXecutable Contracts (X-contracts, Section A.5, pg. 255), a similar language targeted at the expression of business contracts, benefit from a publicly available definition of their syntax. Clearly, whatever other qualities these languages may have, it would not be feasible to adopt them as the basis for specifying SLAs with genuine financial implications, as the parties to these SLAs would have no strong basis for arguing for any particular interpretation of the SLAs in the event of a disagreement.

BCL, RBSLA and X-contracts are principally concerned with the expression of rules in a similar manner to, or explicitly based on, deontic logic. Deontic logic allows the statement of permissions and obligations for parties to perform various actions [137]. This emphasis tends to create a language that makes it easy to describe the protocols by which interact parties are bound, and results in useful SLA terminology in situations where the risk is primarily related to violations of this protocol. For example, the failure to deliver a good following the submission of a purchase order may result in an obligation to pay a penalty.

However, it is not clear how easy it is use such semantics as a basis for precisely describing more quantitative conditions, such as a reliability condition limiting the number of failed service requests within a sliding window. In addition to describing relationships between boolean propositions, such as a violation implying the obligation to pay a penalty, it is also necessary to describe judgements over more complicated domains, such as whether a set of events, with properties represented by strings or numerical values, represents a violation.

The highly-expressive combination of EMOF and OCL, used to define SLAng and its extensions, supports this in a very understandable way, thanks to its reliance on object orientation, and can also represent permissions and obligations implicitly by associating violations with the history of monitored events, and explicitly by asserting in definitive documentation that the violation of a constraint represents the violation of a permission or obligation pertaining to a party. BCL and RBSLA tend to obscure the semantics of complex judgements by relying on external definitions of non-primitive events (e.g. the violation of a latency constraint), or external ontologies of metrics. X-contracts, which have a representation and semantics based on finite-state machines are likely to require unfeasibly complicated statements to represent conditions pertaining to a large amount of service history, due to the state-explosion problem.

These contract languages do highlight the advantages of a more restricted formal underpinning in work relating to the validation of contracts: for example, checking for conflicting obligations, or asserting liveness properties of the protocols being described. EMOF and OCL, by contrast, allow testing of these properties. Testing typically does not offer such strong guarantees as validation that relies on model checking. However, it may be applicable to more types of quality attribute. A possible future enhancement for SLAng may be to define a sub-language for the expression of permissions and obligations, to make conditions that are naturally expressed in this manner more amenable to automated validation.

X-contracts are the only prior work to consider monitorability in any sense. The authors recommend that a middleware supporting non-repudiable message exchange be used to monitor contracts. This would make it impossible to deny violations related only to positive actions (for example, violations of prohibitions). However, I do not believe it solves monitorability issues related to obligations or temporal constraints, because of the difficulty of attributing the cause of delays or faults to the action of a single party in the case where interaction is with multiple remote parties, for example, both an electronic- and network-service provider.

To the best of my knowledge, SLAng is unique in providing support for accuracy constraints in relation to the gathering of evidence. SLAng's support for, and emphasis on conditions relating to the termination of SLAs is also novel, reflecting a general lack of understanding that the duration of a service-provisioning relationship may be a major risk factor for the parties involved. However, both requirements are mentioned in early work related to BCL, but not elaborated upon in later work describing the language.

Older related work is principally concerned with describing QoS for various types of electronic service interface. This includes OWL-S (Section A.8, pg. 259), QML (Section A.9, pg. 259), and QuO-

QDL (Section A.10, pg. 262). This work relies on the implicit assumption that service providers can be trusted to describe the quality-of-service provided by their own services, and then deliver services to that level. Based on this, the focus of these works is first to develop vocabularies of useful metrics, then to consider how these specifications can be used to compose services with predictable QoS characteristics. In my view this work relies on a fundamentally unrealistic assumption, and the languages are unsuitable for describing SLAs since they do not allow the specification of penalties and hence the mitigation of risk. Moreover, the definitions of the metrics are typically informal, hindering reliable analysis. QML, for example, allows the matching of contracts according to user-defined orderings of metrics, which, as discussed in [119] can result in matches that are less safe than my notion of compatibility for SLAs.

The QuA project (Section A.11, pg. 262) has described an approach to formalising the semantics of SLAs that was somewhat influential in the decision to base the semantics of SLAng on a model of service behaviour.

I also surveyed JSDL (Section A.11, pg. 262), a language for specifying parameters for jobs to be executed on a computational grid, discussed previously in relation to the case-study. The information specified using JSDL impacts upon the QoS properties of the job execution, so the comparison with SLA languages is relevant. JSDL suffers from the flaw that it allows the client to specify how quality should be delivered, rather than what is required, which is unfairly restrictive of the service provider. This approach is not appropriate for SLAs between financially independent parties, and is unlikely to be appropriate even for computation grids where the grid provider and the client are financially independent.

Finally, I examined the use of property values in trading services such as CORBA trading, and UDDI. The trading services do not ascribe any semantics to the properties that may be expressed, and again rely on the good auspices of the service provider to ensure that the service levels implicitly guaranteed are delivered. However, the flexibility of these services means that a language like SLAng could be used to specify commodity SLAs when advertising service offerings.

In summary, all of the languages reviewed suffered from serious deficiencies with respect to my requirements. My principle complaints against all of these languages are, that in comparison to SLAng they are not particularly helpful in expressing the conditions that need to be included in ASP SLAs. The languages do not define what is needed, or provide a significant contribution or guidance towards the production of necessary extensions. Also, having an SLA drafted in one of these languages inspires no real confidence that it can be used as a mechanism to mitigate risk, because the SLA will be imprecise – largely due to imprecision in the definition of the language, and a lack of traceability between the SLA and the specification of the language in which it is defined – or because the conditions included in the SLA cannot be monitored – most frequently because they related to the state of a computer system that cannot directly be observed. In comparison, SLAng meets these requirements to a large extent.

It may be that in some cases the usefulness of these languages could be radically improved with relatively little effort. In particular, BCL and X-contracts both seem to have contributions to make with respect to modelling business protocols. However, in neither case has a syntactic definition of the language been made available. Details of the languages can only be inferred from descriptions and

examples published in academic papers.

8.3 The power, adequacy and specificity of SLAng

I now consider in more detail the extent to which SLAng meets the requirements for power and restrictiveness in SLA languages, in so far as these are measured by the metrics for power, adequacy and specificity that I defined in Section 4.5.

The metrics defined in Section 4.5 all depend on a size metric, which can be regarded as a mapping from language specifications or models to the domain of real numbers, and also a function that maps a model to those elements of its language specification upon which it depends. In Section 4.5.3 I described how these functions could be defined for languages based on EMOF and OCL with embedded comments in natural language, the approach used in SLAng.

The size metric developed in Section 4.5.3 had a number of parameters that control the weight given to various features in a model. I now investigate what choices of parameter values are suitable for measuring SLAng and SLAs expressed using SLAng.

instanceWeight	referenceWeight	enumerationWeight	nullWeight	integerWeight	booleanWeight	realWeight	stringWeight	stringElementWeight	size()
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3030.0
0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	10520.0
0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	705.0
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	42875.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	866.0
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	966.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	12.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	3325.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	162001.0
1.0	1.0	1.0	0.0	1.0	1.0	1.0	0.0	0.0	16099.0
1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0	0.0	19424.0
1.0	1.0	1.0	0.0	1.0	1.0	1.0	0.0	0.1	32299.1

Table 8.1: Various measures of the size of the SLAng specification

Table 8.1 shows various measurements of the size of the SLAng specification, based on different choices of parameter values. The first nine lines all represent measurements in which the weight of a single kind of feature has been set to 1, and all other kinds set to zero. This provides a set of counts of the features of various kinds in specification.

In order for measures of power, specificity and adequacy to reflect the balance of effort in producing a language, its extensions and a model, it is necessary to apply the same size metric to all three elements when calculating the measurements. It is also preferable for the size metric to correctly measure the proportion of effort devoted to defining each type of feature in a model or language specification, as there

is no guarantee that the relative proportions of each type of feature will be the same across languages, models and extensions. Therefore a bias in weighting the features could result in a bias in the calculation of the derived metrics for power, specificity and adequacy.

Determining the relative efforts involved in specifying different types of features is difficult, and an empirical study would be required to determine values with any degree of confidence. However, based on the measurements of individual feature types for SLAng, it is possible to apply some "rules of thumb".

Nulls in a model or language can represent a conscious decision to omit some information. However, $42,875$ nulls are present in the SLAng specification, in comparison to $10,520$ object references and only $16,394$ attribute values of all kinds (the sum of reference, enumeration, integer, boolean, real and string counts). Therefore, weighting nulls equally to other values would disproportionately skew the size metric towards specifications with large number of optional attributes. In fact, the high number of nulls in the SLAng specification is due to the structure of the OCL metamodel, in which the various syntactic types potentially have several different possible types of owners in the syntax tree, only one of which is ever actually present. Due to the large number of OCL expressions and sub-expressions in SLAng, this results in a lot of nulls, but the choice of super-expression for a sub-expression can't really be regarded as a major decision in the specification process (the reverse decision contributes to the the reference count).

The other overwhelmingly large size measurement is the count of the characters in all strings included in the specification. The vast majority of these characters are in natural-language comments embedded in the specification. These characters are important in explaining the specification, and definitively establishing the semantics of the domain model. However, they are clearly not an order of magnitude more important than all other types of feature put together. However, perhaps they are equally important.

In light of these considerations I have opted for a combination of parameters that assigns no importance to nulls or to the number of discrete strings. It weights instances, references, enumeration values, integers and reals as 1. It weights characters in strings as 0.1, thus providing a measure for the size of SLAng of $32,299.1$, around half of which is contributed by characters in strings, and the rest by the structure of the specification, and the non-null primitive values (apart from strings) that it includes.

In Table 8.2, I present more size measurements for various sub-components of the SLAng language, and the language extensions and SLAs produced in the case-study. Because the weighted size metric potentially obscures the relative importance of strings and structural elements, I also present measurements of the number of strings, characters, and other types of features, for each measured artifact.

These size measurements provide a number of useful insights. Most significantly, despite the importance of the domain model for improving the precision of SLAng, the definition of the syntax of the language is more than five times larger. This difference in size is even more pronounced in the extensions required for SLA 1 and SLA 4 which, combined with the high specificity of these extensions (measured below), suggests that the domain model provided by SLAng is highly adequate.

This is a useful observation because at present it is not clear how the contribution of the domain

Artifact	Non-string elements	Strings	Characters	Weighted
SLAng syntax	12321.0	2486.0	113803.0	23701.3
SLAng semantics	2010.0	457.0	24710.0	4481.0
SLAng types	1504.0	328.0	11187.0	2622.7
SLAng generic syntax	6783.0	1369.0	58143.0	12597.3
SLAng electronic-service syntax	5538.0	1117.0	55660.0	11104.0
SLA 1 extensions	9345.0	1931.0	88249.0	18169.9
SLA 1 syntactic extensions	8865.0	1815.0	82081.0	17073.1
SLA 1 semantic extensions	476.0	114.0	5992.0	1075.2
SLA 1 generic syntax extensions	4599.0	950.0	39387.0	8537.7
SLA 1 electronic-service syntax extensions	4266.0	865.0	42694.0	8535.4
SLA 1	905.0	163.0	6091.0	1514.1
SLA 4 extensions	6553.0	1356.0	58129.0	12365.9
SLA 4 semantic extensions	19.0	8.0	502.0	69.2
SLA 4 syntax extensions	6530.0	1346.0	57451.0	12275.1
SLA 4 generic syntax extensions	4405.0	913.0	38406.0	8245.6
SLA 4 electronic-service extensions	2125.0	433.0	19045.0	4029.5
SLA 4	370.0	58.0	1126.0	482.6

Table 8.2: Sizes for various sub-components of the SLAng language, and language extensions and SLAs produced in the case-study

model to the power of the language can be calculated (the `used` function proposed in Section 4.5.3 only identifies relevant syntactic types). However, because the domain model is relatively small, it will not greatly influence the metrics. In the measurements below I calculate specificity both including and excluding the semantic types. The measured specificity is lower when including the semantic types, because the measurement implicitly assumes that these types are not relevant to the SLA. However, it is clear that these types are at least somewhat relevant to the SLA, as they serve as the foundation for the semantics of the language in which it is defined. The specificity measurements therefore define a range within which the true specificity of the language (or extension) must reside.

I now present the results of power, adequacy and specificity measurements of SLAng plus the extensions defined for SLA 1 to the expression of SLA 1:

- Power of SLAng plus SLA 1 extensions for SLA 1: 0.966
- Adequacy of SLAng to SLA 1: 0.604
- Specificity of SLAng syntax to SLA 1: 0.990
- Specificity of SLA 1 syntactic extensions to SLA 1 : 1.00
- Specificity of SLAng syntax plus extended syntax to SLA 1: 0.994
- Specificity of SLAng overall to SLA 1: 0.847
- Specificity of SLA 1 extensions overall to SLA 1: 0.943
- Specificity of SLAng plus extensions overall to SLA 1: 0.873

The fairly obvious conclusions to be derived from these values are that the definition of SLAng plus the extensions needed for SLA 1 is overwhelmingly larger than SLA 1 itself, so taken together SLAng

plus the extensions represents an extremely powerful language for expressing SLA 1. However, SLAng required significant extensions to support the expression of SLA 1, only contributing around 60% of the specification requirements. Regardless of whether semantic elements are considered, SLAng and the extensions are highly specific to the expression of the SLA. SLAng syntax only fails to be 100% specific due to the inclusion in the specification of the concrete class `InformalFailureModeDefinition`, which turns out to be not useful to SLA 1. This insight suggests that specificity measurements are useful for identifying the accidental inclusion of inessential elements when developing domain-specific language support.

A similar set of values were obtained for SLA 4:

- Power of SLAng plus SLA 4 extensions for SLA 4: 0.987
- Adequacy of SLAng to SLA 4: 0.682
- Specificity of SLAng syntax to SLA 4: 0.990
- Specificity of SLA 4 syntactic extensions to SLA 4: 0.993
- Specificity of SLAng syntax plus extended syntax to SLA 4: 0.991
- Specificity of SLAng overall to SLA 4: 0.847
- Specificity of SLA 4 extensions overall to SLA 4: 0.990
- Specificity of SLAng plus extensions overall to SLA 4: 0.879

In the case of SLA 4, SLAng plus the extension were again highly powerful, and SLAng was somewhat more adequate to this SLA. This reflects the fact that the electronic-service to which conditions in SLA 4 were related was a simple, stateless, synchronous webservice, and the conditions required were more straightforward than for SLA 1. SLA 4 therefore required a smaller extension to the core language. Here the specificity measurements for the syntax of the extension also reveal the inclusion of a superfluous class `FixedDeadlineTerminationByReportReconciliationAdministrationClause`.

In summary, an approach to defining SLAs using SLAng as a basis results in highly concise SLAs in comparison to the overall information burden of the SLA. The remaining information is encoded in SLAng, and extensions to the language. In the case-study, SLAng provided more than half of the specification requirements for both SLAs.

The fact that the specificity of the SLAng syntax to both SLAs is high indicates that almost all of the core SLAng syntactic definition is used in some way in both SLAs, suggesting that the design of SLAng correctly anticipated the expressiveness requirements for these two SLAs, and contains little that is superfluous. Combined with constraints in the language to prevent the expression of bad SLAs, this result contributes to the assessment that SLAng is highly restrictive, a good property in a domain-specific language, as it reduces the effort required to construct a correct statement.

The adequacy of the core SLAng language to the case-study SLAs was not as good as might be desired (strictly speaking, 100% adequacy is always desirable; however, this is not necessarily compatible with producing a powerful and restrictive language). However, based on my survey of alternative languages, summarised in the previous section, I contend that SLAng is probably more adequate to these SLAs than any other alternative language. This assertion may be supported by the observation that although the SLAng core language contains very few concrete classes, specificity is still high. This indicates that almost all of the core SLAng classes are being used in extensions that are subsequently being used in SLAs. SLAng contains sophisticated support for conditions such as timeliness, reliability, and throughput which are either omitted or not specified with the same degree of precision in previous languages, in addition to support for conditions relating to termination and measurement precision that are universally absent from other languages. Clearly to express the same conditions using one of the alternative languages surveyed, support for these elements would have to be included in a language extension, as it is absent from the cores of these languages. I would therefore expect these languages to be less adequate to the expression of these SLAs.

In the next section, I discuss how future evolution of SLAng could improve its adequacy to new SLAs.

8.4 A trajectory for SLAng

In the previous section I concluded that SLAng is potentially the basis for powerful languages and has good specificity for ASP SLAs, as demonstrated by the SLAs prepared for the case-study. However, the adequacy of SLAng was not as good as might be desired, with both case-study SLAs requiring extensions that took a significant amount of effort to produce, as indicated by their size relative to the core SLAng language specification.

As indicated by the high specificity of the core SLAng language to both case-study SLAs, up to this point SLAng has been designed to express only that which is essential to all ASP SLAs. However, there is potentially plenty of SLA syntax which is necessary for many SLAs, but not all. If this were incorporated into the core language, then the expected amount of support provided for any new SLA would increase, at the risk of incorporating support for statements that will not in fact be required in a particular SLA. In other words, the average adequacy of the language can be improved at the expense of the average specificity. However, this approach must not be taken to extremes, as this will result in a bloated language specification that is consequently difficult to understand and automate.

In this section, I demonstrate that under some circumstances it is possible to obtain large gains in the average adequacy of a language, in return for relatively small decreases in average specificity.

The measure taken to seek to improve the adequacy of SLAng to SLAs 1 and 4, while maintaining a high specificity is very simple. I produced a language specification in which all extension elements common to both SLA 1 and SLA 4 are combined. This is the language specification that forms Appendix E. By regarding the combined extensions as part of the core language, the adequacy of this core is clearly increased. Moreover, because the extension elements are required by both SLA 1 and SLA 4 (the original extensions for which were highly specific), specificity of this core language does not decrease.

Artifact	Non-string elements	Strings	Characters	Weighted
Combined extensions	4888.0	1020.0	44472.0	9335.2
SLA 1 extensions	4468.0	919.0	44466.0	8914.6
SLA 4 extensions	1632.0	331.0	13575.0	2989.5

Table 8.3: New sizes of the language extensions for the case-study SLAs after common elements are combined

The new sizes of the extensions, and the size of the combined extension elements are shown in Table 8.3.

The power, adequacy and specificity measurements obtained for the SLAs expressed using the combined language, plus residual extensions are as follows:

- Power of SLAng, with combined extensions, plus SLA 1 extensions to SLA 1: 0.966
- Adequacy of SLAng plus combined extensions to SLA 1: 0.818
- Specificity of SLAng syntax plus combined syntactic extensions to SLA 1: 0.993
- Power of SLAng, with combined extensions, plus SLA 4 extensions to SLA 4: 0.987
- Adequacy of SLAng plus combined extensions to SLA 4: 0.923
- Specificity of SLAng syntax plus combined syntactic extensions to SLA 4: 0.993

As expected, the figures show an increase in adequacy, without a corresponding drop in specificity. Naturally, the decision to incorporate new syntax into a language must be based on the requirement to support the expression of a known set of statements. The set of statements consisting of SLA 1 and SLA 4 is a small one on which to base this decision. However, evolution of the language must necessarily proceed based on the assumption that future SLAs will tend to resemble those already encountered to some degree, and as further work generates a greater corpus of SLAs on which to base language design decisions, these decisions can become more sophisticated.

The work described in this section highlights the usefulness of my metrics as a tool for guiding the evolution of a domain-specific language such as SLAng, which has open-ended expressivity requirements. It also suggests a future trajectory for the design of SLAng, in which future design decisions are informed by the experience of conducting more case-studies, but regulated by the evaluation of my metrics over a corpus of SLAs developed across all of the case-studies.

8.5 Summary

In this chapter I have documented three exercises intended to shed light on the value of SLAng as the basis for defining ASP SLAs. In the first, I considered in detail the extent to which SLAng conforms to its requirements. I found SLAng to meet the requirements to a high degree, except in three areas: expressiveness, because extensions are required to the language to define SLAs; feasibility, because the formulation of conditions in current versions of the language is not amenable to efficient interpretation; and analysability, because the semantics of the language are not aligned with the models required by known analysis techniques.

In the second exercise, I compared SLAng to alternative languages with the potential to express SLAs, and found it to be superior to all alternative languages in almost all categories defined by the requirements. Where some alternative languages had better properties than SLAng, it was related to analysability. However, these properties also harmed the precision of these languages for expressing the intent of the parties to the SLA precisely. In all other respects, SLAng was better suited to defining ASP SLAs. Despite the fact that it does not provide full support for any particular SLA, it still appears to be more adequate than previous languages in the support provided for the types of conditions commonly needed in the scenario, such as timeliness, reliability and throughput, and also other necessary elements absent from other languages, such as conditions related to termination and measurement accuracy. Also, it is more precise than previous languages, in that it is both formally defined, and traceability can be maintained between SLAs and the definition of the language. Finally, the emphasis placed on monitorability of the language means that the SLAs expressed will be mutually-monitorable, with accuracy constraints approximately-monitorable by both parties to each agreement.

In the third exercise, I considered in more detail the expressiveness of SLAng, by taking measurements of the language, and the language extensions and SLAs defined in the case-study. These demonstrated that defining SLAs based on SLAng resulted in concise SLAs compared with the size of the language specification, and that the language specification was highly specific to the task of defining the SLAs in the case-study. However, large extensions to the language were needed to define the extensions, confirming the earlier assessment that the language was not as expressive as might be desired, an inevitable consequence of the conflicting requirements for restrictiveness and expressiveness in ASP SLAs.

In a continuation to this final exercise, I demonstrated how the experience of applying SLAng to the definition of SLAs could be used to refine the design of the language in order to increase its adequacy, by incorporating syntax from extensions found to be useful in multiple SLAs. Specificity measurements may be used to control this process in order to prevent the language specification from becoming bloated and hence unusable.

Chapter 9

Summary

In this chapter I summarise the research contributions made by the work presented in this dissertation, offer some final perspectives concerning the work, and present some initial ideas concerning future research continuing from the work.

9.1 Contributions of this work

Faced with the opportunity to outsource part of their business using an application service, it is natural for a party to contemplate entering into an SLA, whether negotiated or fixed by the service provider. The author of any SLA is likely to seek support for the production of the SLA in the form of a domain-specific language.

This work has presented the design and implementation of a language for ASP SLAs, SLAng. SLAng is intended to meet a set of requirements that I enumerated in Chapter 2. These requirements were based on a number of assumptions, also introduced in that chapter. In my view, the two most important assumptions made were: first, that the principal role of an SLA is to mitigate for the client the financial risks implied by an outsourcing relationship; and second, that the domain of services under consideration were application services, which are characterised by the use of electronic services for the majority of communications between the parties.

The first assumption, combined with an explicit lack of assumptions concerning the worth of the service to the client, and the trustworthiness of the parties involved in the service provision relationship, has led to a focus in this work on the protectability of SLAs, by which I mean those qualities of an SLA that increase the likelihood that any dispute pertinent to the SLA will be resolved in accordance with the original intent of the parties. I have looked at two important contributions that a language can make towards the protectability of the SLAs that are written using it: first, in terms of the precision with which SLAs may be expressed; and secondly, in terms of restrictions on the SLAs that may be expressed to ensure that the conditions that they include are monitorable. A precise, mutually-monitorable SLA will be protectable, because in the event of a dispute, the original intent of the SLA can be retrieved from its concrete representation, and used to produce an unambiguous judgement in relation to a set of evidence concerning the service. The trustworthiness of the evidence will be apparent to the parties to the agreement, because they will have had the opportunity to directly observe the events from which it is derived, or have evidence concerning the events provided to them by a party that they trust, and which is not financially interested in the service-provision scenario.

The requirement for precision in SLA languages led me to propose the adoption of a meta-modelling approach to the definition of SLA languages, which requires a language specification to include object-oriented models of both the syntax of the language being defined, and the semantic domain to which its statements pertain. The semantics of the language are precisely defined by relationships between these models, expressed in terms of ordinary object-oriented relationships, refined through the use of a logical constraint language. The approach I described is based on the standard languages EMOF and OCL, with HUTN and XMI standards contributing concrete syntices for the language.

Because a concrete document represents the record of an SLA, and because the meaning of that document depends on the meaning of the language in which it is expressed, the traceability between statements and language specifications has an impact on the precision of an SLA. If the traceability does not exists, or is ambiguous, the association of the SLA with the language can be disputed, and hence the original intent of the SLA can be disputed. I considered the provisions for such traceability in the standards upon which my language-specification approach is based. As a result, I proposed improvements to the standards that guaranteed traceability between statements, concrete syntax standards, formal descriptions of syntax and semantics, and any natural language documentation essential for definitively establishing the semantics of a language. I implemented tool support for these proposals, discussed how this tool support could be useful in testing a language, and its potential as a component in a system for monitoring compliance to SLAs.

The requirement for SLAs to be monitorable to the highest degree possible led to my formalisation of the notion of monitorability as an abstract mathematical model. I described and demonstrated how this model could be permuted in order to identify systems of SLAs with particular monitorability properties, and which could act in concert to insure risks to parties in a service-provisioning scenario.

Measurements of events are inevitably subject to error. It would be impractical for an SLA to require a party to monitor events perfectly, therefore a constraint is required on the measurements used when calculating violations that permits some error, but not too much. This constraint would preferably be monitorable by all parties, but this is impossible because no party can know the true values of the events being monitored with certainty, and so determine whether another party is making an intolerable number of errors. Therefore, the constraint should be approximately monitorable, so that a party can assess whether the constraint is being violated with greater than a certain probability. I described the design of such a constraint, and demonstrated that it was approximately monitorable using a statistical hypothesis test.

The second assumption upon which this work rests, that electronic services are a characteristic component of ASP scenarios, permitted the identification of typical service-provision roles and infrastructure in the services for which SLAs were to be specified. Usually, electronic service interactions consist of messages passed between some client software and some server software, operating on nodes distributed in one or more networks. The behaviour of the client software is typically the responsibility of a client party, the behaviour of the server software is the responsibility of a service-provider party, and the behaviour of the network may be the responsibility of a separate network-service-provider party. All

three parties may be financially independent.

This model serves as the foundation for two major contributions of this work: first, by analysing the monitorability of systems of SLAs applied to the ASP scenario, I was able to demonstrate that only a single system existed in which the risk to the client implied by the possibility of delayed responses from the service was mitigated, all SLAs were mutually-monitorable, and in which the use of SLAs implied no new financial risks to the parties that are not also mitigated. In this system, the network-service provider offers an SLA to the client concerning the events constituting the service as it is perceived at the client's interface to the network. The service provider then offers the network-service provider a compatible SLA concerning events at the service-providers interface to the network.

This result is highly significant. It implies either that more work is needed into producing trusted monitoring solutions, providing parties the opportunity to monitor events in a trustworthy way at locations in the network to which they do not normally have access, or that network service providers will have to start to act as service resellers, a business model not in common use today, or that mutually-monitorable SLAs will not be practical in the ASP scenario. Since monitorability is such a desirable property for SLAs, in this work I rejected the final possibility, and assumed the second. As a consequence of this, it was possible to scope the vocabulary for my SLA language at an interface, or electronic-service, level of abstraction, rather than having to consider QoS in networks.

The second contribution resulting from the assumption of the ASP model also follows from the monitorability result. Having chosen to produce a language for mutually-monitorable SLAs, in which conditions on electronic services apply only to events occurring at network interfaces, it was possible to incorporate into SLAng support for the expression of those conditions that the scenario implied would commonly be required, designed so as to be mutually-monitorable. In my initial discussion of the scenario, I identified three such types of condition: latency, reliability and throughput. I also observed that to mitigate all major risks in the scenario, the parties may wish to associate penalties with premature termination of the SLA, and the provider will need to use the SLA to charge for the service.

In my initial requirements, I highlighted the need for expressiveness in an SLA language, a consequence of the need to express the highly various factors that may affect the magnitude of the risk to which a party is exposed, and hence imply a need to vary the effect of SLA conditions. However, I also introduced requirements for restrictiveness, to assist an SLA author in specifying SLAs meeting their requirements to a high degree, and power, so that the language can capture as much domain knowledge as possible in a reusable manner, therefore reducing the amount of information that needs to be specified in an SLA. Requirements for expressiveness, on the one hand, and power and restrictiveness, on the other, appear to be contradictory. To address this issue, I proposed that an SLA language should be specified in an abstract, extensible manner, and described how this could be achieved using the meta-modelling approach that I mandate for SLA languages. The example language developed in this work, SLAng, was implemented in this manner. The consequence of this choice is that usually extensions to SLAng must be defined before it is possible to fully specify a desired SLA.

The thesis in question in this work concerns whether this language represents an improvement over

previous languages designed for the same purpose, particularly in terms of the protectability of the SLAs that can be expressed using SLAng, and the level of support provided for the expression of the conditions required in the SLAs. The evaluation of SLAng in these terms presented some further challenges due to the highly subjective nature of the judgements involved.

In order to demonstrate the practicality of SLAng, I used it to specify SLAs appropriate to an actual service provisioning scenario, in a case-study. I treated the case-study as an exercise in which a new technology was introduced into a service-provisioning scenario in an attempt to meet outstanding requirements. Since SLAs are concerned with mitigating risks, the case-study included a risk analysis applied to the scenario. I was then able to apply the result of my monitorability analysis of electronic-service provision to guide the high-level design of a system of SLAs capable of mitigating the identified risks (except for security risks, which I have ruled outside the scope of this work), and all new risks implied by the use of SLAs. I subsequently demonstrated that SLAng could be used as the basis for an implementation of these SLAs, resulting in two fully specified SLAs, and the language extensions upon which they depend.

Based on the experience of the case-study, I next evaluated SLAng against the my requirements for SLA languages and language specifications. I also compared SLAng to a broad survey of previous SLA languages, evaluated in the same terms, and found it to be superior in both the level of support provided by the language for expressing conditions appropriate to the ASP scenario, and the protectability of the SLAs produced.

In order to further investigate the contribution provided by SLAng to expressing ASP SLAs, I proposed a set of metrics for extensible domain-specific languages. ‘Power’ measurements attempt to assess the distribution of effort between defining the language elements used by a statement, and defining the statement itself. ‘Adequacy’ measurements assess the relative contributions made by a core language and its extensions to expressing a statement, and ‘specificity’ measurements act as a control on power and adequacy measurements, discouraging the production of bloated language specifications by measuring the relative sizes of used and unused language definition elements with respect to a statement. Having discussed how these metrics may be defined for languages defined using EMOF and OCL, I applied them to measuring SLAng, and the SLAs and extensions developed during the case-study.

I found SLAng, combined with the extensions required to specify the SLAs in the case-study, to be highly powerful and specific, with respect to the case-study SLAs. This suggests that the language is quite restrictive, correctly implements the required support, and that the cost of producing SLAs may be low in comparison to the cost of producing the language. These are good features for the language to have, as they suggest that the language can usefully be reused, and that effort spent specifying the language will be saved when specifying SLAs using the language.

However, the extensions developed in the case-study were somewhat large compared to the language itself, increasing the effort required to specify the SLAs. I therefore demonstrated how SLAng could be evolved to incorporate commonly required syntactic support, identified as a result of experience using the language. I combined the extension elements required by both the case-study SLAs into

a common extension package, and demonstrated that if this package were regarded as part of the core language, then the adequacy of the language for the SLAs was significantly improved, without compromising its specificity. I suggested that the principled incorporation of such extensions, controlled by the taking of measurements according to my metrics, represents a promising future trajectory for SLAng.

9.2 Conclusions

The principal focus of the work presented in this dissertation has been SLAng, a language for ASP SLAs, which I have evaluated as meeting a range of requirements for ASP SLA language to a greater extent than had been previously achieved.

However, I do not consider SLAng in its current form to be a magic bullet, capable of enabling the immediate widespread adoption of SLAs in ASP, precipitating a revolutionary increase in the degree of outsourcing in which enterprises engage, thereby delivering all of the increases in quality and efficiency promised by exponents of that model. This is for two reasons: first, I do not consider SLAng to be the last word in SLA languages for ASP; and second, that language support alone cannot address all of the challenges that must be overcome in order to use SLAs successfully.

Making use of SLAng currently involves defining extensions to the language using EMOF and OCL, requiring expertise which, at the time of writing, is not widely available. I believe that to be broadly practical, the adequacy of SLAng must be further improved in the manner described in Section 8.4, a gradual incorporation of syntax that has been found to be useful in practical applications of the language. Eventually, SLAng might include ready-made syntax such that in many cases no extensions are required, or the parties will prefer to slightly modify their intended agreement in order that it can be conveniently expressed in SLAng.

SLAng is also immature. In this work I have identified the capabilities that EMOF and OCL support for testing a language specification, but a thorough approach to validating a language in this fashion remains the topic for future work. Without such validation, little confidence can be had that the formal elements of SLAng truly reflect the design intent described in this document, and embedded in informal comments in the language specification. Similarly, revision of the language must be considered to improve the feasibility of automatically checking conditions expressed in the language.

The peripheral challenges surrounding the definition and use of SLAs in ASP scenarios also remain considerable. This was demonstrated in the case-study, in which although it was possible to identify the types of risks to which scenario participants are exposed, it was difficult to determine the parameters or detailed design for SLAs capable of correctly mitigating these risks. This suggests that alongside future work in developing languages for ASP SLAs, methodological studies are also required to develop approaches to quantifying financial risks experienced by parties in ASP scenarios. The effect of SLAs on these risks must also be better understood. This includes a more detailed consideration of the exploitability of SLAs.

In this work I discussed the concept of monitorability, and discovered an interesting result in relation to the ASP scenario, which was that to achieve mutual-monitorability in a safe system of SLAs, network providers potentially have to resell services and monitor the provision of these services at the point

where they are delivered to their clients. Clearly, no insurmountable theoretical barriers prevent this kind of monitoring. However, the practical implications of this result are tremendous, in that it would require network service providers to become much more involved in the business of ASP, and to perform monitoring of a kind that they currently do not. This would inevitably imply increased costs, at least in terms of initial investments in monitoring infrastructure. Naturally, there will be considerable resistance to this, and it is likely to be a roadblock to the adoption of this kind of SLA.

However, the requirement for monitorability is based on the assumption that trust between the parties does not exist. If these trust relationships, which can broadly be summarised as the expectation that the parties have that other parties will honour their commitments, can be better quantified, then it may be possible to relax the requirement for monitorability. Alternatively, it may be that future research into cryptographic techniques, or trusted-computing platforms, can deliver trusted monitoring solutions that will enable parties to obtain monitoring data from remote locations with a high confidence in its voracity. This would potentially allow the monitoring of end-to-end QoS properties, essential for monitoring network services at a network-level of abstraction, and also allow the network service providers to export monitoring into the infrastructure of their clients, rather than implementing it in the network.

Despite these future challenges, I believe that this work has made a significant contribution to the state of knowledge concerning the role of SLAs in ASP, and the use of domain-specific languages to support the authoring of these SLAs. I have found issues relating to ASP SLAs to be surprisingly complex and multi-dimensional, involving considerations of risk, finance, trust, measurement theory, and the contribution of domain-specific languages to the qualities of the statements that they express. The need to confidently address these issues as a prerequisite to using SLAs seems to convincingly explain the current lack of adoption of what seems to be a promising complementary technology to conventional electronic-service middleware. I have found that previous work tends to neglect important considerations in these categories, and the work documented here therefore represents some first steps in mapping these issues and the contribution that DSLs can make to addressing them.

In the next section I discuss future work in more detail.

9.3 Future work

9.3.1 On domain-specific languages

Model-Driven Engineering (MDE) and the OMG's Model-Driven Architecture (MDA) are currently hot topics for research. Approaches based on models tend to emphasise the production of domain-specific languages or language extensions, and the production of SLAng can be seen to be an example of this kind of activity.

As discussed in Chapter 4, using a lot of different languages in any enterprise, including software development, has advantages and disadvantages. The advantages are delivered by the power and restrictiveness of the languages. The disadvantages are due to the need to have expertise in each of the languages used, and to integrate information expressed in numerous different language. I justified my recommendations for embedding natural-language documentation in language specifications, and for preserving traceability between statements and the languages in which they are written, with reference to

these problems. The metrics I developed for the measurement of certain properties of languages and language extensions are also intended to be useful MDE settings, as language development will inevitably need to be the subject of quality control, which can benefit from quantitative support.

However, MDE is not a mature field, and further work needs to be done on the fundamental technological basis for such developments. The most important technologies required are a meta-modelling language and related tools, capable of supporting the definition of both the syntax and semantics of languages, the management and processing of statements in languages, and the integration of information for statements in different languages. The OMG is making a concerted effort to define such a language and technologies, with its standards for MOF, OCL, JMI and the Query-View-Transformation (QVT) language [92], which is intended for data integration. Based on the experience gained implementing some of these standards for this work, there are problems with them, the most important of which are: there are too many of them; their integration is not well understood; and they are too complicated.

These problems were manifested to some extent in this work, and the design of SLAng. As discussed in Chapter 3, there is no true conceptual difference between expressing an instance of an object using a concrete syntax such as HUTN, or XMI, and describing a MOF class which can only have a single instance in the context of the model in which it resides. Therefore, it seems that these standards could be combined, by simply extending EMOF to permit the expression of instances, or rather, classes containing only properties with constant values. This would eliminate the apparently unnecessary distinctions between instance specifications (which can describe multiple objects, one in each of several situations conforming to a statement) and classes (which can describe multiple objects within a situation conforming to a model), models and meta-models, and power and adequacy measurements. This would also enable the engineering of model repositories that do not need to be recompiled when new meta-theories are added. My tool support for calculating metric values, which incorporates the MOF semantic model into the meta-model for the repository, represents a first demonstration of such functionality.

A similar duality exists between OCL and QVT, both of which essentially only describe relationships between model elements. As evidenced by experiments on an early version of SLAng, it can be hard to write constraints in OCL that evaluate efficiently. Moreover, OCL cannot specify new instances of objects as a result of expression evaluation, making certain calculations difficult to specify in an object-oriented manner that is both clear and amenable to evaluation. QVT, with its imperative constructs can do both these things, but relies on OCL to define invariant relationships between models, hence resulting in a duplication of specification effort. Preferable would be to have a declarative language for consistency that could be operationalised to perform model transformations, provided relationships are described with a sufficient lack of ambiguity. In the context of these issues, further consideration must also be given to the suitability of OCL as a language for describing constraints that can efficiently be evaluated, as discussed in Section 4.4.2.

Revolutionary changes to the underlying standard languages aside, more work is also needed to advance the methodological aspects of model-driven engineering. When developing SLAng, I settled on the idea of using an abstract, extensible language to address the trade-off between requirements for

expressiveness and requirements for power and restrictiveness. This could be regarded as being an MDE design pattern. It may be profitable to document it as such, and consider other design patterns that may be useful in the development of DSLs.

Further work is also need to develop more sophisticated metrics for measuring the usefulness of metrics for models and languages. In addition to power and adequacy, a measure of the degree of structural guidance provided by an extensible language to developers would be desirable, which might be called *extensibility*, and measure the reliance that an extension has on abstract types and operations defined in the core language.

In this work, I have tended to rely on textual notions for my language specification and models. However, the role of diagrams in MDE developments is still extremely important. As mentioned in Section 4.1.3, future work is needed to determine the best way to integrate this kind of documentation into language specifications and models.

9.3.2 On risk

Apart from understanding what qualities a good SLA should have and knowing how to specify one, issues which I have addressed to some extent in this dissertation, the other major impediment to actually entering an SLA is knowing what parameters the SLA should include. This requires knowledge of two things: first, the kinds and magnitudes of the risks to which the parties to the SLA are exposed; and second, the effect of SLA conditions on these risks.

Obtaining the first kind of information requires an effort of analysis. In the case-study, I was able to identify the kinds of risks to which the parties were exposed by considering clearly negative outcomes of the various activities included in use-cases for the system. However, this approach, although appealing from common-sense perspective, has not been subject to validation, and it would be desirable to demonstrate that it was genuinely a thorough approach to analysing risks, or find a better alternative. Also, the approach did not quantify the risks, partly due to a lack of availability of financial information. It clearly needs refinement.

Under some circumstances there may also be a predictive aspect to this kind of analysis. As discussed in Chapter 2, the decision to outsource and hence enter into an SLA will often be made before the investment in integrating an outsourced service. Hence, determining the magnitude of some risks may also mean making financial and development predictions. Some interesting work at predicting quantitative aspects of software-development projects using Bayesian belief networks conditioned using genuine historical data is described in [26]. It would be interesting to consider how this approach could be extended to include outsourcing decisions.

Clearly, the semantics of the SLA language in which an SLA is defined are relevant to determining the effect of the SLA on the risks to which scenario participants are exposed. If, as described in the previous section, the feasibility of checking conditions using the semantics of the language could be improved, the language specification and SLA have the potential to be used directly in simulations of service behaviour, in an attempt to predict cash flows under various assumptions concerning the behaviour of participants and the service.

9.3.3 On trust and monitorability

As mentioned above, trusted monitoring platforms could modify the set of respondents to an event, and hence the potential to specify highly-monitorable SLAs for a given service situation. Most desirable would be the possibility to specify SLAs that could be arbitrated by a third party that is both mutually trusted by both the client and provider, and also able to obtain trustworthy evidence concerning all events pertinent to the SLA. Trusted monitoring platforms would also allow network-service providers to export monitoring from the network infrastructure into client and service-provider infrastructure, eliminating the costs of deploying new monitoring infrastructure for the sole purpose of supporting monitorable SLAs.

Several challenges must be addressed when considering trusted monitoring. These include the need for monitors and their output to be tamper-proof, as they may be executing on infrastructure controlled by a party with a financial incentive to cheat. The link between a trusted monitor and the consumer of its output must similarly be secure. For most applications, trusted monitors will need access to a trusted source of time-stamp information.

I have also mentioned the interaction between trust and monitorability in the presence, rather than the absence, of trust between interacting parties. The SLAs described by SLAng imply a rather high monitoring burden. It would be interesting to considered, theoretically, how quantified levels of trust between parties could be used to reduce this burden using statistical sampling.

Finally, monitorability has an effect on trust, in that a party can trust a fact if they have observed it themselves. It would interesting to integrate a well-known model of trust relationships with a model of monitorability, to observe how confidence in the exchange of information can be established when certain parties can verify certain facts by observation, and may or may not also be known to be able to do so.

9.3.4 On SLAng

Although SLAng exists primarily as an exemplar for the theoretical innovations introduced in this work, it is not inconceivable that it could be used as the starting point for a process of refinement and augmentation that could eventually result in a broadly useful ASP SLA language. I demonstrated the potential of my metrics to assist with such development in Section 8.4. In order to achieve this, the experience of defining SLAs in a number of realistic scenarios will be required. One possible approach to obtaining such input would be to propose SLAng, or a similar language, for adoption as a standard by some industry body. The language could therefore benefit from input from any participating organisations. I would expect that the language would need to evolve through several versions before reaching a broadly satisfactory level of maturity, in a similar manner to OMG standards such as UML.

As mentioned above, the formal aspects of SLAng will also require validation to generate confidence that they capture the design intent for the language. Testing is a possibility, but consideration will need to be given as to how to guarantee coverage. It may be possible to redesign SLAng so that certain aspects of its semantics, such as constraints over the payment of penalties, are aligned to a known formalism, such as deontic logic, thereby enabling validation by model checking.

It is possible that time will show that the monitorability assumptions built into SLAng are too stringent for realistic use. However, precision is a useful property, even for SLAs that are not intrinsically monitorable. It may therefore ultimately become useful to incorporate additional syntax pertaining to unmonitorable conditions. Conversely, the development of trusted monitoring systems may eventually make arbitratable SLAs possible, in which case SLAng could profitably be extended to enable the specification of this type of SLA.

The application of SLAs to risks related to the security of services have not been considered in this work. However, requirements to do so in SLAs have been identified in previous work [74], and a risk that could plausibly be mitigated by an SLA, related to confidentiality, was discovered in the case-study scenario. A survey of security risks needs to be conducted to differentiate between those to which SLAs may contribute, and those that are better addressed by service-implementation technologies, such as access-control systems or secure communications. Where SLAs are an appropriate mechanism to mitigate a security risk, consideration must then be given to the design and formalisation of conditions to do so, and the monitorability of these conditions.

Precise SLAs are potentially of relevance to other technical domains. Work in the TAPAS project highlighted the need to formalise component and application hosting relationships [49]. Such relationships represent a formidable challenge for trusted monitoring. However, assuming that this challenge could be overcome, it would be helpful to develop SLAs capable of describing the resources provided to a hosted component over time.

Another important area in which a role for SLAs has been identified is the provision of help-desk services [126], possibly as an adjunct to an application service. Availability conditions supported by SLAng, related to the exchange of bug and bug-fix reports (which are essentially support activities), hint at the possibility of formalising conditions relating to the provision of these services.

Appendix A

Critical review of alternative languages for ASP SLAs

A number of languages for SLAs appropriate to the ASP model have been proposed in the academic literature, and publicised by industrial organisations. No language has yet achieved broad adoption in this area, although WS-Agreement is the product of a standardisation effort by the Open Grid Forum (OGF). Some academic work also exists that attempts to establish principles or requirements for ASP SLAs without contributing to the definition of any particular language. It is also the case that a number of distributed systems technologies exist that permit the use of service meta-data that could be considered to be SLAs.

In this appendix I review work in these categories. Of these, the most complete and recent are languages for SLAs for web services. Also significant are older attempts to define SLAs for CORBA platforms.

A.1 The Web-Service Level Agreement language (WSLA)

WSLA is a language for web-service SLAs developed by IBM [34]. The language is specified in a technical report [34]. The syntax of the language is defined using an XML schema [24], with semantics described using natural language.

A SLA written in WSLA consists of a set of preliminary definitions, establishing who the parties to the service are, and what service is being constrained. It then defines 'service level parameters' and a set of 'service level objectives'. The parameters, described using 'measurement directives', identify quantities to be measured for a service, and provide the opportunity for parties to specify how a quantity should be measured, who has responsibility for monitoring it, and from where measurement data can be retrieved. Service objectives are constraints over measured quantities expressed in a typed-expression sub-language consisting of various functions and predicates that may be combined hierarchically.

In addition to defining service objectives, WSLA SLAs can define obligations on parties to take particular types of action under given conditions, including the violation of service objectives.

The WSLA specification is separated into a core language, a set of standard extensions, and acknowledges the need for user-specified extensions in addition. WSLA relies on the extensibility of XML schemas to support this.

To what extent does WSLA provide support for expressing conditions to mitigate the risks involved in the ASP scenario?

WSLA provides no explicit support for expressing reliability, and throughput constraints, or constraints on the real-world behaviour of the service. However, it does include a framework of abstract schema types from which these conditions could straightforwardly be extended, and also the definitions of some measurable quantities of a service, such as response times and invocation counts. WSLA includes no explicit support for assigning financial penalties to parties in response to condition violations, or indicating that an SLA should terminate. It does contain a limited expression language for expressing conditions over measured values or functions of these values, although this is not expressive enough for a formal definition of reliability or throughput in terms of primitive events. The violations of such conditions can trigger actions, the only given example of which is notification. However, new action types may be defined in extensions, so financial penalties could feasibly be defined. WSLA does not discuss payment schemes, or exploitability.

How do SLAs expressed using WSLA contribute to increasing the likelihood that a dispute concerning an SLA will be resolved according to the original agreement?

The WSLA specification is quite understandable. SLAs written in WSLA are XML documents, and are superficially easy to understand. However, they are very imprecise. The semantics of WSLA are expressed in natural language. Moreover, the most important element of an SLA, the definition of conditions, is highly reliant on the definition of measurement directives using schema extensions. The only property of these extensions required by the abstract data-type that they must extend, is that measurements have a defined data-type. This leaves room for significant ambiguity in how the measurements should be obtained. An example directive given for response time only defines a name and a data type. Nothing is specified concerning what events contribute to defining response time, so disagreements are possible concerning whether response time should be measured at the client's interface or the service provider's. Additional imprecision is introduced by a lack of prescription concerning traceability between SLAs and the specification, and the duality of the schema provided for the language, which is not definitive, and the PDF specification, which, I assume, is.

No consideration of monitorability or error is given in the specification, although support is given for specifying who should obtain a given measurement and from where (expressed as a web-service port from which monitoring data can be obtained). In several places it is suggested that measurements may be obtained by polling. This is extremely risky from a monitorability perspective because polling fails to measure the behaviour of the service that is of genuine concern to a party, instead electing to measure a different behaviour that assumed to be an indicator of required quality. Polling a system for performance generally requires an identifiable form of request that has no adverse side-effects, as it should be expected that the poll has no business semantics. Polling request can hence be distinguished from real requests. It is therefore possible for any party to cheat the intent of conditions over polling requests by reserving capacity.

How does the design of WSLA and its language specification contribute to reducing the costs of outsourcing activities, including the authoring of SLAs?

WSLA is somewhat restrictive in that it enforces syntactic rules and provides some minimal guid-

ance for the producers of specification extensions. It is not highly powerful or adequate, in that it defines little in the way of semantics, so the burden of expressing the meaning of an SLA is largely passed to the author of the SLAs and any required extensions. The schema for the language is automatable to check the validity of WSLA SLAs, but the lack of formal semantics hinders further automation and analysis. Exploitability of WSLA SLAs is not discussed.

A.2 The Web-Services Offerings Language (WSOL)

WSOL is primarily the work of Vladimir Tosic and Kruti Patel, in post-graduate work at Carleton University, Ottawa, Canada. Details of the language are revealed in a series of academic publications [132, 105, 134, 133].

The authors do not claim that the language expresses SLAs, but instead 'web-service offerings'. An offering is a specification of both the performance of a service, including constraints over timeliness and reliability metrics, and its interface, achieved by reference to a WSDL document. Offering hence form a more complete specification of the behaviour of the service than WSDL can provide alone.

The distinction between offerings and SLAs for web-services is largely artificial. Offerings are characterised as being authored by the service provider, allowing them to differentiate several pricing schemes for the same service, and there is the implication that the advertisement and selection of services based on offerings could be highly automated. However, in practice machine-readable SLAs could perform the same role, with providers offering commoditized contracts. Conversely, nothing prevents the parties from negotiating the terms of a web-service offering.

Service offerings are similar to WSLA service objectives. They employ a limited expression language to describe a condition on a number of 'metrics' over the behaviour of the domain, defined in an external ontology. They identify an accounting party with the responsibility for monitoring the offering. They can also include or refer to management statements that describe actions to be undertaken in the event of a violation of the offering.

To what extent does WSOL provide support for expressing conditions to mitigate the risks involved in the ASP scenario?

Like WSLA, WSOL relies on extension to provide details of the properties of services to be measured in relation to conditions. The authors recommend that these be provided in 'external ontologies' for QoS metrics, measurement units, measured properties (of a service), measurement metrics, and currency units. By way of an example, the need to specify the following information in a definition of a QoS metric is asserted: a name; a textual description; links to the ontology of measured properties; formulae by which given QoS metric can be computed from other QoS metrics; and invariant relationships with other QoS metrics. Unfortunately no specific requirements are listed for the other ontology types required. Nor is a prescription made concerning the manner in which the ontologies should be expressed. A review of existing web ontologies provided by the authors reveals previous work to be inadequate with respect to the requirements stated, but no example of an adequate ontology for metrics is provided. In [105] an example ontology for measurement units is defined using XML conforming to a simple schema for ontologies. The ontology simply lists the well-known measurement unit names

'millisecond', 'second', 'minute' and 'hour' without additional documentation.

In light of this highly inadequate treatment of the fundamental properties to which conditions must relate, it is hard to assert that WSOL provides any real support for mitigating the risks involved in ASP. WSOL does provide some support for charging schemes and assigning penalties. In the documented syntax for the language the following types are included: pay-per-use price statements; subscription price statements; monetary penalty statements; and management responsibility statements. Termination of agreements is not considered, nor is exploitability.

How do SLAs expressed in WSOL contribute to increasing the likelihood that a dispute concerning an SLA will be resolved according to the original agreement?

WSOL suffers from same problems as WSLA. It is largely defined using natural language, and has no documented principles for managing its extensions. These problems are seriously compounded by the lack of definitive documentation for the language (assuring a lack of traceability between SLAs and their meanings), and the inadequate provision of ontologies defining major parts of the meaning of the language.

Third parties can be assigned management responsibilities in WSOL service offerings, which may include monitoring. However, no discussion is provided concerning monitorability in scenarios in which some parties may not be trustworthy. The monitorability of an offering would depend on metrics and measured properties defined in an external ontology. Monitorability is not discussed in the requirements for such ontologies, and nor is measurement error.

How does the design of WSOL and its language specification contribute to reducing the costs of outsourcing activities, including the authoring of SLAs?

Like WSLA, WSOL inherits usability benefits from its reliance on XML schema to define its syntax. In addition, considerable efforts have been spent by the authors to make service-offering specifications reusable via various mechanisms. These include an extension mechanism for service offerings, the ability to cross reference declarations made elsewhere in a number of contexts, and a template system for parameterising several types of syntax. It is not clear why these are required. WSOL would be a powerful language if it had an adequate semantic definition – much of the semantic burden of an offering would be assumed by the documentation of the language and any external ontologies used. Offerings should therefore be concise. It is also not clear whether this type of reusability is useful in a language for expressing documents to which confidentiality requirements may apply.

The lack of clearly defined ontologies or standards for these ontologies for WSOL is clearly a barrier to analysability and automatability.

A.3 Web-Services Management Language (WSML)

WSML is described in a technical note published by Hewlett Packard [111]. The objective of WSML is to express SLAs for application-services, and it is acknowledged that this may include real-world behaviour as well as constraints on electronic services. Like WSLA and WSOL its syntax is defined using an XML schema. Also like WSLA it requires schema extensions to provide concrete types for certain necessary elements in an SLA.

The design of WSML places its main emphasis on two properties of the language, *flexibility* and *precision*. The report in which the language is described states that flexibility is necessary because of the broad variety of requirements that a client may have. The requirement for precision in WSML is stated differently from my use of the term. For the authors of WSML 'precision' connotes a complete, categorised description of the requirements of the parties. Hence the authors state that every service-level objective in an SLA should specify the following information:

- A time constraint on when the clause applies
- What physical quantity is being measured.
- When the condition should be evaluated.
- What subset of measurements is relevant to calculate the condition.
- What function is used to calculate whether the condition holds.
- What action should be taken if the condition is violated.

Each of these categories of information is represented by an abstract syntactical type which can be extended to allow the specification of whatever details are pertinent to a particular SLA. This perspective was influential in the design of SLAng when considering how classes should define abstract operations to guide extensions.

To what extent does WSML provide support for expressing conditions to mitigate the true risks involved in the ASP scenario?

Like WSLA a set of concrete semantic extensions are in the technical note defining WSML, although no claim is made for the completeness or usefulness of these. Instead they serve as examples to support the claim that WSML can express any contents that users could desire in an SLA. Example measurement functions include determining whether a service has been available for a percentage of some period, and determining whether response time is greater than some threshold. No discussion is made of reliability, or of how the correct functional behaviour of the system could be documented or referred to in an SLA. However, an extension to the language could in principle address this issue. No example of the constraint of real-world behaviour is given, although the author's assert the capabilities of the language in this respect.

No example is provided of the specification of actions in response to service-level objectives being violated. To mitigate financial risk in a predictable manner an extension describing the payment of penalties would be needed. Termination of SLAs is not discussed, nor is exploitability.

How do SLAs expressed in WSML contribute to increasing the likelihood that a dispute concerning an SLA will be resolved according to the original agreement?

In a nod to the notion that the semantics of SLAs should be formally defined, the report states that the function whereby violations are calculated should be specified in a mathematical notation such as MathML [140]. However, no example of this is provided in the report.

In other respects, WSML suffers from the same defects as WSLA in terms of understandability. Its semantics are defined in natural language, and it relies on extension of its specification without defining how those extensions should be managed or documented.

Although not explicitly stated as a requirement for extensions, the examples given make explicit use of XML schema documentation elements, which represents a responsible, if informal approach to preserving the meaning of the language in situ with syntactic definitions.

However, with respect to the core language, it is not clear whether the report in which the language is documented should be considered normative documentation of the language for any reason except pragmatically as no other documentation is publicly available.

The structuring of service level objectives around the ontology for such definition discussed above does support a responsible approach to defining SLAs. In particular the language is superior to WSLA in its insistence that the quantity being measured, as opposed to the means by which measurements are obtained, be specified.

The designers of WSLA rightly think it suitable to specify constraints in relation to other properties of an electronic service than solely the responses received over the network, for example the state of availability of the service. However, the authors rely on the implicit, and in my view unrealistic assumption that anything that may be constrained in an SLA is either monitorable or the party responsible for the quantity can be trusted to report on it. No treatment of error is provided in the description of the language or provided for in its syntax

How does the design of WSML and its language specification contribute to reducing the costs of outsourcing activities, including the authoring of SLAs?

Like WSLA and WSOL, WSML accrues benefits related to its reliance on XML schema. WSML is not a highly adequate language, but once extensions have been defined, it is reasonably powerful. A realistic example SLA given in the report is 70 lines of XML, although this SLA cannot be said to meet my requirements to a high degree.

The informal semantics of WSML do not intrinsically support analysis, and no discussion of analysis is provided. Nothing prevents the expression of exploitable SLAs in WSML. Exploitability is not discussed in any work related to the language.

A.4 Rule-Based Service-Level Agreement language (RBSLA)

RBSLA is a language for the specification of SLAs for electronic services, designed according to the (undischarged) assumption that it is preferable to base the semantics of an SLA language on logic programming [103] rather than any other approach. It extends the standard rule language RuleML [110] with a number of new concepts, which the author asserts are useful for expressing SLAs. RuleML is a language for rules based on the Resource-Description Framework standard (RDF) [144], a language for describing resources, their properties, their relationships, and their types, which is in turn based on XML and XML schemas. The extension proposed for RBSLA are: typed logic with types and modes; procedural attachments; external data integration; ECA rules with sensing, monitoring and effecting; (situated) update primitives; complex event processing and state changes (fluents); deontic norms including vio-

lations and exceptions; defeasible rules and rule priorities; built-ins, aggregate and compare operators, and lists; additional compact if-then-else-syntax; SLA-domain-specific elements such as metrics, escalation levels and ontology-based domain-specific contract vocabularies; and test cases for verification and validation of SLA specifications.

To what extent does RBSLA provide support for expressing conditions to mitigate the true risks involved in the ASP scenario?

It is difficult to assess the true level of support provided by RBSLA, due to the lack of a comprehensive specification document. A complex XML schema for the language is available, but it does not appear to provide any vocabulary particularly related to reliability, timeliness or throughput conditions. The semantics of the language are clearly being developed incrementally. [104] provides an account of the semantics for the event-condition-action extensions to RuleML, which clearly support part of the RBSLA proposal. [103] indicates that, similarly to WSOL, SLA-domain specific elements will be provided by external ontologies, and suggests the use of OWL ontologies [143] or Java class-hierarchies. However, a documented example of this has not yet been made available.

How do SLAs expressed in RBSLA contribute to increasing the likelihood that a dispute concerning an SLA will be resolved according to the original agreement?

RBSLA clearly represents an attempt to place the specification of SLAs on a highly formal basis. However, the understandability of the language is seriously undermined by the assumption of a massive, but incompletely-documented, formal apparatus supporting the semantics of the language. Contrast this with the relatively simple object-oriented domain theory provided by EMOF. Because of the barrier to comprehension that it imposes, the use of such apparatus must be justified in terms of the kinds of proofs of correctness and consistency that can be provided for the language or the SLAs that it is used to express. However, this is not adequately motivated in discussions of the language.

How does the design of RBSLA and its language specification contribute to reducing the costs of outsourcing activities, including the authoring of SLAs?

RBSLA does not seem to be a particularly powerful domain-specific language. Instead, it focusses on highly flexible syntax supporting a number of different types of rules concerning the conduct of parties and services. These rules appear to be a light syntactic sugaring of what is, in essence, a logic programming language, so RBSLA more closely fulfils the role of EMOF than SLAng. Although using RBSLA may be easier than expressing an SLA in Prolog, it still seems likely to require a high degree of expertise by the author. This may be an acceptable trade-off in that it may result in SLAs that are highly amenable to analysis (such as proofs of correctness) and automation. However, the precise contribution of the formal underpinnings of the language to the practical aspects of service management have yet to be demonstrated.

A.5 EXecutable Contracts (X-Contracts)

X-contracts are a proposed representation for contracts related to interactions between electronic services, principally aimed at enabling monitoring and enforcement, where ‘enforcement’ although not explicitly defined, appears to connote the automation of contractual obligations [71]. The principal

recommendation of the approach is that bilateral contracts be represented using pairs of Finite State Machines (FSMs), one each for the client and provider parties. Transitions in the machines are labelled with events, which correspond to actions the parties have a right to perform, and to external circumstances such as timeouts, and with outputs, which represent the enactment of obligations. Shared events cause the co-evolution of the state of both machines over the duration of the relationship between the parties.

X-contracts may be used to specify SLAs. However, the main focus of the work is on supporting the specification of protocols by which the parties to an agreement must abide. These protocols may include temporal constraints, and hence touch upon issues of quality-of-service.

To what extent do X-contracts provide support for expressing conditions to mitigate the true risks involved in the ASP scenario?

FSMs are a useful formalism for describing protocols. X-contracts therefore represent a reasonable approach to representing agreements in situations where risks are primarily related to adherence to protocols. [70] includes a realistic example in which a purchase order and invoice must be exchanged according to a loose schedule. Failure to adhere to this protocol implies the obligation to pay penalties. On the other hand, it is not clear how adequate FSMs are as a formalism to specify the desired behaviour of a service in terms of the functional relationship between its inputs and outputs, or its aggregate performance over a large number of usages (which would tend to require a large FSM to maintain a history). The X-contract approach also does not provide any reusable support for defining common requirements, a flaw that the authors acknowledge, and suggest could be rectified by the construction of a contract template database. X-contracts are not discussed in relation to defining constraints on real-world behaviours, although it is possible that these could be represented as events.

How do X-contracts contribute to increasing the likelihood that a dispute concerning an SLA will be resolved according to the original agreement?

X-contracts have the potential to be extremely precise. Thus far, this potential does not appear to have been completely realised. The FSM model used could easily be associated with a formal semantic definition; however, the authors have not yet provided this, or even a definitive syntax for representing X-contracts. Also, although a graphical notation is demonstrated for representing FSMs, it is not clear to what standard this conforms. In [124], the authors encode an X-contract using the language Promela, which does have a formally defined syntax and semantics, but it is not clear whether this approach should always be used to specify X-contracts, or whether it is the result of applying an unspecified translation from some other representation of an X-contract. The need to precisely define the meaning of events and actions referred to in X-contracts is not considered, nor is the need to specify standards relating to the measurement of the time of occurrence of events.

Some consideration is given to monitorability in work relating to X-contracts. In particular, in [71] the use of the B2Bobject middleware is proposed to mediate interactions [15]. This middleware abstracts interactions as a shared stateful object. Updates to the object are non-repudiable, hence bad-behaviour related to positive action by a party will be monitorable. However, the B2Bobject middleware cannot guarantee timestamp values for events. Nor can it demonstrate whether the absence of an event was due

to a network delay or a protocol violation by a participating party.

How do the recommendations relating to X-contracts contribute to reducing the costs of outsourcing activities, including the authoring of SLAs?

FSMs are not particularly restrictive or powerful. However, they are highly analysable and automatable. In [124] the authors verify a Promela specification of an X-contract against various requirements using the model checker, SPIN. Implicit in the notion of enforcement frequently referred to in relation to X-contracts is the potential to base the implementation of a service on the automated enactment of its X-contract obligations. Likewise, automated monitoring using the B2Bobject middleware is enabled by the simulation of an X-contract at runtime.

A.6 Web-Services Agreement Specification (WS-Agreement)

The WS-Agreement specification [100] has the distinction of being the only proposed standard supporting the representation of SLAs for electronic services. It is under review by the Open Grid Forum (OGF) for standardisation.

In common with several languages for the same purpose, WS-Agreement provides an extensible XML Schema. Abstract elements that can be made concrete in order to support the definition of a particular SLA are: identifiers for the client and provider of the service; references to related agreements; service descriptions; and guarantee terms. A service description must have a name. A guarantee term must: reference the service in relation to which the guarantee is being made; list some domain-specific variables to which the guarantee pertains; optionally express a precondition under which the guarantee holds; express the condition that must be met to satisfy the guarantee; and assign a 'business value' that associates a priority, and a penalty or reward with the guarantee. Penalty definitions may include the definition of an assessment interval, a quantity or expression defining the penalty, and a unit.

In addition to this syntactic framework, the standard specifies a service and life-cycle for the creation and management of agreements using a web-services. According to this life-cycle, an agreement factory maintains a list of agreement templates, which are parameterised agreements together with a set of prerequisites for entering the agreement. A client contacts the factory with an agreement offer, which satisfies the prerequisites and sets the parameters for the agreement. This triggers the instantiation of two new webservices, one to represent the agreement, and the other a client-specific port onto the service that was originally requested. The service modelling the agreement provides access to agreement details, can indicate the state of the underlying service, and indicates whether guarantees in the agreement are currently being violated.

To what extent does WS-Agreement provide support for expressing conditions to mitigate the true risks involved in the ASP scenario?

The extent to which WS-Agreement provides support for specifying agreements is described in full above. WS-Agreement provides no intrinsic support for expressing reliability, latency and throughput conditions. Indeed, due to the informal and high-level way in which the semantics of the language are specified, and the extremely abstract nature of the language, it is debatable whether WS-Agreement should be regarded as a language at all, rather than a vague description of the gross properties that an

SLA should possess.

The definitions of penalties in WS-Agreement acknowledge the need to consider issues of administration and finance in SLAs. However, they are too vaguely defined to be an adequate basis for an agreement as to these elements without further extension.

How do SLAs expressed in WS-Agreement contribute to increasing the likelihood that a dispute concerning an SLA will be resolved according to the original agreement?

Since WS-Agreement defines no semantics specific enough to determine whether violations of an agreement are being met, it cannot be regarded to contribute anything towards guaranteeing the production of protectable SLAs. This will entirely be the contribution of extensions to the language.

The lifecycle described by the WS-Agreement standard is also highly dubious from a monitorability perspective. Clearly the service-provider would implement the agreement factory, and therefore the webservice representing the agreement. The implication seems therefore to be that the service provider will monitor the service on behalf of the client. Even assuming the service-provider could be trusted to do this honestly, they will not necessarily have access to the point of service provision of interest to the client, namely the client's own interface to the network.

How does the design of WS-Agreement and its language specification contribute to reducing the costs of outsourcing activities, including the authoring of SLAs?

Again, the sparsity of syntactic definitions and lack of actionable semantics for the language make it impossible to say that it contributes usefully to the production of SLAs. Although it is intended that a monitoring service should be instantiated automatically as a result of entering into a WS-Agreement, this will necessarily require knowledge of the semantics of terms in the agreement, none of which are specified by the WS-Agreement standard.

A.7 The Business Contract Language (BCL)

BCL is a language for describing contracts consisting of sets of deontic obligation, described in a number of research publications [55, 29, 74].

In BCL a contract is a reification of the concept of a community taken from to RM-ODP specification [36]. BCL defines community types. A community type has a set of roles, action templates, policies, events, states, sub-communities and instantiation rules, which govern when a community conforming to the type is deemed to exist. Roles identify parties and objects in abstract. Action templates define types of actions in which objects may participate, and have action roles. Constraints may be applied to roles. Policies are constraints with deontic modes: permissions, prohibitions and obligations. Conditional policies may be modelled by the exchange of tokens: permits and burdens. Communities can be hierarchically combined by regarding the community as an object whose overt behaviour can satisfy the conditions on a role. Events may be primitive or emergent. Event matching constraints and event patterns bridge the gap between primitive events and non-primitive events and actions. State occupancy may related to events and referred to in constraints. State may be represented by a combination of variables with various types.

The author's assert that BCL has an XML representation, and also a non-standard human-usable

notation, and further that tools supports exists for deserialising these notations [55]. However, neither tools nor definitions of these representations are available in the public domain.

To what extent does BCL provide support for expressing conditions to mitigate the true risks involved in the ASP scenario?

From the example provided in [55] it seems that BCL primarily provides a type theory for contracts, in other words, asserting that everything of interest to a contract can be described using one of the syntactic types defined above. The events, actions and policies for a contract must then be implemented in a particular contract. It seems that, similarly to RBSLA, BCL is aiming to provide a richer, but more complicated and restrictive, meta-theory for SLAs than that provided by EMOF. BCL does not seem to contain domain-specific support for defining reliability, latency or throughput constraints for electronic services. The need to consider termination conditions is mentioned in the requirements stated for the language, as is the payment of penalties [74], and an example of the obligation to pay a penalty is included in [55].

How do SLAs expressed in BCL contribute to increasing the likelihood that a dispute concerning an SLA will be resolved according to the original agreement?

Clearly the intention for BCL is to provide a precise semantic for the language, based in part on deontic logic. In [29] a mapping from BCL policy statements to a logic is described in order to detect conflicting obligations or prohibitions. However, this mapping is partial and not definitive of the semantics of the language. Moreover, the unavailability of a complete description of the syntax of the language makes it hard to assess its expressive capabilities.

Monitorability is not considered. Also, the example of an SLA provided in [55] includes a definition of availability based on events related to the transition of a webservice from an operational to inoperative state, an event that is unlikely to be fully monitorable even to the provider of a web-service, still less the client. The treatment of error is also not considered, although is mentioned in an earlier paper concerning the requirements for BCL [74].

How does the design of BCL and its language specification contribute to reducing the costs of outsourcing activities, including the authoring of SLAs?

BCL does not seem to be a highly powerful language, in that it does not succinctly describe contracts by reusing known constraint types. However, the structuring of the syntax described by the authors does suggest that the language would be restrictive. However, BCL will contribute little towards defining SLAs or any other contract until a sufficiently complete and unambiguous description of its syntax and semantics is published.

A.8 Ontology Web Language for Services (OWL-S)

OWL [143] is a language for Web ontologies, based on the RDF. OWL is a cornerstone of the Semantic Web initiative. It is a powerful language for defining ontologies, which are in turn vocabularies for expressing meta-data about resources on the web. The intent of the Semantic Web initiative is that internet resources should be extensively labelled with meta-data using such vocabularies in order to support a high level of automated reasoning about, and manipulation of these resources. This could

include more intelligent search algorithms than are currently possible, and automated composition of services.

OWL-S [146], formerly known as DAML-S, is an OWL ontology for web services. It allows the specification of three types of meta-data concerning services:

- A service profile.
- The process by which the service should be used.
- The grounding of a service.

The OWL-S specification states that the service profile may be used in service discovery, and that it can specify functional and non-functional properties for the system. The specification of functional properties is supported by the definition of meta-data properties for services such as pre- and post-conditions. The specification of non-functional properties relies on the definition of ontologies external to OWL-S, with no specific examples given.

In its intent to describe the properties that a web-service has, without involving the client in any negotionation, OWL-S is similar to WSOL. It would require extensions to the specification of truly bilateral agreements between parties.

OWL has a highly formal semantics closely related to description logics that supports inferences over meta-data descriptions. However, these inferences are largely to do with categorisation, and it is not clear that a QoS ontology expressed using OWL would be any more valid, or less ambiguous in its relationship to the service domain than less structured approaches. In contrast SLAng, with its explicit service model provides a direct description of the meaning of measurement data and conditions over that data. In principle, an ontology of SLAng constraints could perhaps be described in OWL with reference to the SLAng specification, allowing the reuse of SLAng statements in meta-data compatible with OWL-S.

A.9 Quality-of-service Modelling Language (QML)

QML is a well-defined language specified in a technical report from Hewlett Packard. It's intent is quality-of-service description, rather than the expression of SLAs, although this is clearly an important component of an SLA language. Despite predating WSML, also from HP, there is little resemblance between the languages.

The syntax of QML is defined using a traditional BNF grammar. It is machine readable, but not based on XML. Instead it resembles an interface definition language, such as CORBA IDL [90], with which it is intended to be complementary.

In contrast to the languages previously reviewed, QML does not rely on language extension to provide flexibility. Despite the emphasis placed on modelling QoS rather than expressing SLAs, QML relies heavily on the concept of a QoS contract. It allows the definition of contract types, which consist of sets of *dimensions*. Each dimension is a user defined QoS property, with a type. Numeric dimension types may be given a user defined order, which indicates whether larger or smaller values of the type are considered to deliver better performance.

Contracts are instantiations of contract types, in which each of the dimensions is associated with a constraint. For numeric types this includes the expression of standard relationships that must be true for the values in the dimension, and also constraints on statistics of the value, such as the mean and variance.

Finally contracts may be associated with an appropriate interface element, such as an operation, with the interpretation that quality attributes of the element conforming to dimensions in the contract must conform to the constraints specified in the contract. The association is called a *profile* for the element, and in the sense that an element may have multiple profiles is similar to WSOLs notion of a service offering.

QML provides no syntax relating to penalties or actions to be performed in the event of an element failing to conform to a profile with which it is associated. I therefore do not consider it an SLA language, and will forego detailed comparison with our requirements.

Another major deficiency in the language relates to the ability of users to define custom contract and dimension types. Definitions of these types only identify the mathematical structure of the domains of these types, and give them a name, without allowing further description of what instances of the type are supposed to correspond to in the real world. Whilst the type system makes the language highly extensible the meaning of the individual metrics in the context of the software system is not formally established. Since exchange of SLAs between parties requires a common understanding of such metrics, this can be regarded as a serious deficit.

QML does define a rigourous semantic for its contracts and profiles, but this definition is not based upon their real-world interpretation. Instead a denotational semantic describes how contract declarations imply the existence of elements in a mathematical contract domain. Axioms then allow deductions concerning the typing of these elements.

Additional axioms define a relationship called conformance that pertains between contracts and also between profiles. Conformance is discussed further below in the section on SLA analysis. A contract is said to conform to another if it is compatible with the type of the second and its constraints are stronger according to the user-defined ordering on their dimensions. This allows comparison of contracts. However, discisions made on the basis of these comparisons may not be safe due the the disconnection between the semantics and the real-world systems described.

The emphasis on typing is clearly an attempt to make the language flexible, without relying on extensions to the language definition, therefore permitting the assertion that the language in its published form is adequate to the expression of any SLA. However, unlike programming languages that incorporate similar type facilities, an SLA language cannot rely on a semantic definition consisting of a few primitive notions, constructed by types and data structures into more complex behaviours, because it may have to refer to things in the real world not anticipated by the language designer. Consequently, the language must either allow the embedding of extra documentation in contracts, or be extensible. QML supports neither of these options, and so is semantically inadequate to the expression of SLAs.

QML has been reused in the architecture description language CBabel [3] to specify QoS contracts using infrastructure dependent dimensions prior to the deployment of a system.

A.10 Quality-of-service for CORBA Objects QoS Description Language (QuO-QDL)

QuO [57] is a CORBA specific framework for QoS adaptation based on proxies. It includes a quality description language, QDL, used for describing QoS states, adaptations and notifications. Part of the language relates to the definition of 'contracts', which in this case are expected relationships between client behaviour and received QoS. Within contracts, 'regions' of QoS behaviour (expressed as constraints over multiple QoS dimensions) are specified, including permissable transitions between regions related to changes in client behaviour and resource availability. The use of the term contract in this context is more closely related to its meaning in contract based programming, where it has the sense of a technical guarantee, than in an SLA where the sense is legalistic.

Dimensions in the language are defined to be the domain of results obtained by invoking instrumentation methods on remote objects. Like WSLA, no formal constraints are placed on the implementation of these methods. In the context of the QuO project, this instrumentation is related to adaptable QoS properties supported by the framework. In this sense the semantics of the dimensions are directly related to the implementation of the service. This is appropriate given that contracts in QuO are supposed to help clients plan their usage, and the system to adapt. However, in the absence of strong trust relationships between clients and service providers, contracts of this form would not be appropriate SLAs. In addition no facilities are provided to describe the consequences of the service deviating from its prescribed regions of QoS.

A.11 Quality-of-service aware component Architecture (QuA)

The QuA project adopts the most rigorous approach to defining the semantics of QoS properties [125], although to my knowledge they have yet to define a concrete syntax for representing SLAs. According to their model, all QoS properties are related to the performance of a service, which supplies a set of operations. Input and output messages are causally related by operation invocations. Output messages are characterised by a set of variables. A set of error functions are defined over the difference vector between an observed output trace for a particular input trace, and the ideal trace as it would be observed were the service deployed on infinitely fast equipment operating without error. SLAs are defined using constraints on the values of error functions.

It is possible to see correspondences between our semantics and the QuA approach. In our case the service model defines the information available concerning service operation, and the OCL constraints provide a concrete representation of the error functions. Features of our semantics not obvious in the QuA approach are constraints independent of a service model assumption, such as the constraint that the service provider must be capable of providing the monitoring solution specified in an ASP SLA, and the ability to constrain client behaviour (in QuA terms, the input trace).

A.12 Quality-of-service Interface Definition Language (QIDL)

QIDL [10] is an extension of CORBA IDL [90]. It does not define constraints on the QoS of a service, but specifies data types and extensions to the functional interface, that are concerned with the monitoring and adaptation of QoS, rather than the direct provision of the service. QIDL allows the implementation

of generic QoS management systems for CORBA services. However, it omits any specification of the protocol according to which this management should proceed to meet the requirements of the client and service provider. It is also not clear that clients in situations where SLAs are required should be offered access to QoS adaptation mechanisms. QIDL is hence not useful for describing SLAs.

A.13 Job Submission Description Language (JSDL)

Like WS-agreement, JSDL is a nascent standard of the OGF. It allows a client submitting a job to a grid-scheduler to specify execution parameters for the job intended to guarantee a certain quality of service [99].

Information in a JSDL job specification can be grouped into essentially four categories:

- Meta-data describing the job: an id, name, annotations and project information.
- Identification of the application/program to be run: At an abstract level a name and version for the application. JSDL also defines POSIX specific extensions, which includes the ability to specify the executable to run, and user and group permissions required.
- A description of the resources that must be made available to the application by the computational resource manager. This information includes names and locations for jobs, required CPU architectures and speeds, memory, operating system and disk partition information, amongst other things.

  This optionally includes specification of: the specific hosts (identified by name) on which the application should be run, any filesystems that should be present, how they should be named, what logical type they should have (normal, swap, temporary or spool), where they should be mounted, show much space should be available to them, whether the application should execute exclusively on its resource, the operating system within which it should be executed, specified by type, name, and version; also quantities of various types of resources, expressed either per-processing node or for the job as a whole: the CPU architecture and speed required, the number of CPUs, CPU time, network bandwidth, virtual memory, and physical memory required.

  POSIX specific extensions allow the specification of limits (representing the upper bound of what may be used and the lower-bound of what is required presumably) on file sizes, core dumps, data segment sizes, locked memory, memory, open file descriptors, pipe sizes, stack sizes, process counts, and thread counts.
- Data staging requirements: These specify required data movements prior and post job execution, and consist of source and target destination specifications.

From this list it can be seen that JSDL is primarily concerned with specifying the means to complete a job successfully according to the client's requirements. Unlike an SLA, it does not attempt to specify the ends required, i.e. what the client's QoS requirements are, or what should occur when these requirements are not met. Since the owners of computational grids and the parties submitting jobs to these grids are likely to be financially independent, the future appropriateness of JSDL seems in question, as

the client must be trusted with assessing what resources are required. This fixes the pricing model of the grid-owner to a reservation-based approach, which may lead to low levels of utilisation in the grid. A results-based pricing model would allow the grid owner more flexibility concerning the allocation of resources.

A.14 SLA information in trading services

The CORBA Trading Object Service [77] allows the advertisement and selection of service offers based on constraints over typed properties. These properties can include QoS specifications, and generally can take any IDL type. Their semantics is not formally defined; neither are external ontologies specified. It is therefore up to the trader and its clients to agree an interpretation for the properties.

UDDI [101] is a directory system for web-services incorporating WSDL descriptions and additional meta-data. It has been criticised for not allowing the specification of quality properties or management information. UDDIe [113] is a proposed extension of the standard that adds additional categories of meta-data, including QoS specifications, to the information provide by UDDI. This QoS information could be regarded as being binding when payment is required for a client to access a service, and so an instance of an SLA. However, the description of eUDDI provides no guidance as to what QoS information should be specified.

It is possible that when using systems such as CORBA trading and UDDIe, which specify the availability of service meta-data, but no strong restrictions on its form or meaning, participants could agree to standardise on a vocabulary for SLAs such as that provided by SLAng.

Appendix B

Case-study material

B.1 Use-case 1: conduct an experiment

B.1.1 Initiating Actor

- Chemist

B.1.2 Preconditions

- Chemist has the necessary access to the `Polymorph Search Webclient`, and has prepared input data files.

B.1.3 Postconditions

- The chemist has completely retrieved the experimental results via the `Polymorph Search Webclient`.

B.1.4 Steps

1. A chemist collects the requirements of the computation. These include application parameters, input and output file requirements, dependencies and system requirements.

   There are no significant risks to any party in this step.

2. The chemist uploads parameters and input files using the `Polymorph Search Webclient`, which is installed on the submission node. The Chemist is interacting with the CS-managed submission node across three networks, that managed by the Chemistry department, the IS network and the CS network.

   Risks to the chemist:

   The chemist will need to engage in a sequence of interactions with the web-client to perform this step. Faults and delays in this process may occur in the Chemist's node, the chemistry network, the IS network, the CS network or the CS node. Therefore, the chemist may be exposed to the following risks:

   Chemistry 1 *Delays in configuring the experiment could significantly delay access by the chemist to the results of the experiment. This could mean missed deadlines, an inability to produce timely results in comparison to researchers from other universities, and similar material harm to the individual chemist and the Chemistry Department.*

Chemistry 2 *Faults may occur at any point in the infrastructure supporting the process. Obvious faults may be temporary, or require intervention by another stakeholder. Non-obvious faults may result in the corruption of the parameters or the input files for the experiment. In theory these faults could occur at any point in the infrastructure. Non-obvious faults become increasingly problematic while they remain undetected. The worst-case scenario in this case is that corruption of data results in the production of invalid scientific results that are relied upon in future by the Chemists.*

It may be assumed that the introduction of non-obvious scientific errors is rare. Data corruption is likely to invalidate files to the extent that the experiment either fails altogether or produces obviously implausible results. Replication of experiments and other validation techniques should trap most residual errors. Nevertheless, such errors may result in a serious amount of wasted time and effort.

Chemistry 3 *The chemistry network must be linked to the IS network and communicate with the web-client. It must therefore be prepared to accept network traffic from the IS network that appears to originate from the CS network. The traffic may potentially act in a manner inconsistent with this behavioural description, either maliciously or accidentally, causing faults or resource exhaustions that reduce the capacity of the Chemistry nodes to perform their usual duties. This is a security risk.*

I cite 'security risk' used in this sense in several different contexts below. Note that such security risks, although normal in today's open networks, are a direct consequence of the need to interact with other networks to communicate and use services. In this case study, I am initially making the conservative assumption that without the motivation to perform computational experiments, the Chemistry department would not necessarily have any need to interact with either IS or CS.

Risks to IS:

IS 1 *Traffic appearing to originate from nodes within the Chemistry and CS networks may be problematic if it acts maliciously. This is a security risk.*

IS must also be aware of the possibility that the communications between Chemistry and CS may be large in volume. Note that Chemistry does not bear an equivalent risk, because it may be assumed that provided risk Chemistry 3 is mitigated, the web-client will function correctly (or Chemistry will receive compensation) and Chemistry should be able to anticipate and therefore deal with the volume of data arriving from CS in response to their service requests.

IS 2 *The IS network must be connected to the Chemistry and CS networks. It must be prepared to accept legitimate traffic from both networks appearing to originate from nodes within those networks. This traffic may be problematic if it appears in too great a volume.*

IS will also need to address the risk that they will not be able to obtain compensation for providing the service.

IS 3 *Conveying legitimate traffic between Chemistry and CS implies a cost to IS. This represents a risk to IS because they may not receive compensation for providing the service.*

Risks to CS:

CS 1 *The CS network must be connected to the IS network. It must be prepared to receive traffic that appears to have originated from nodes in the chemistry department. This may present a security risk.*

CS 2 *Legitimate behaviour may also be problematic if it arrives in too great a volume leading to resource exhaustion. CS servers are a valuable resource, and may not be used exclusively to provide the polymorph search service. Therefore it would be senseless for CS to allow its submission node to be overwhelmed by requests. Webpage requests may reduce the functionality of the* `Polymorph Search Webclient` *if they arrive in too great a volume. The upload of large amounts of data may exhaust the storage capacity of a CS server, rendering it useless for other purposes and clients.*

Like IS, CS will also need to find some way to charge for using the polymorph-search service:

CS 3 *Servicing operation requests submitted to the* `Polymorph Search Webclient` *implies costs for CS due to the provision of network and processing resources. This implies a risk to CS that they will not receive adequate compensation to cover these costs.*

3. The chemist triggers the computation and the web-client acknowledges the start of the computation by displaying an initial status. Again this interaction is with the submission node and occurs across Chemistry, IS and CS networks.

   Risks to the chemist:

   Chemistry 4 *Delays starting the experiment are a risk to the Chemist because the Chemist wastes time waiting for an acknowledgement, and the production of results is also delayed.*

   Chemistry 5 *Faults starting the experiment are a risk to the Chemist because the Chemist may wastes time in the attempt.*

   The risk that an experiment appears to have started but subsequently fails is covered below.

   Network access continues to imply a security risk (Chemistry3).

   Risks to IS:

   Risks are due to opening network to traffic from Chemistry and CS (IS 1, IS 2, IS 3).

Risks to CS:

Granting network and service access to Chemistry nodes continues to imply risks (CS 1, CS 2, CS 3).

4. The web-client triggers the `eMaterials workflow` installed in the BPEL engine on the workflow node. This occurs within CS.

   Risks to the chemist:

   From this point the task of performing the analysis is automated by various parties on behalf of the Chemistry department. Delays and errors occurring in any step of the process can negatively impact upon the chemist. Hence:

   Chemistry 6 *All steps must be completed in a timely fashion for the experiment to be achieved in a timely fashion.*

   Chemistry 7 *All steps must be completed in such a manner as to generate correct results. Hence, data must not be corrupted during transfer, and all components must behave correctly according to this behavioural specification.*

   Risks to CS:

   This step continues to imply risk CS 3, the risk that CS will not be compensated for the service they provide.

   The following two steps only involve action by or between nodes operated by CS:

5. The `eMaterials workflow`, implemented by the BPEL engine, triggers the individual jobs by passing JSDL specifications to a `GridSAM service` instance installed on the submission node.

6. The `GridSAM service` translates the JSDL to a Condor submission file and places it in the Condor submit queue by signalling the `Condor submit daemon`.

   The risks implied by these steps implied are those previously identified to Chemistry (Chemistry 6, Chemistry 7). Also, CS 3.

7. The `Condor controller` node polls the `Condor submit daemon` on the submit node occasionally for information about the queue. When it discovers new submissions, it applies its grid scheduling policy to allocate grid nodes to processing these submissions, informing the submit daemon of the location of the allocated nodes, and each grid node of their allocation to the submission (effectively granting access control of the nodes to the daemon).

   This activity involves communication between nodes operated by CS and cluster nodes operated by IS.

   Risks to CS:

CS 4 *CS servers must communicate with nodes managed by IS. This may constitute a security risk.*

This risk is an additional security risk to (CS 1) as CS servers must now communicate with nodes appearing to be located within the IS network as well as the chemistry network. There is as yet no traffic volume risk to CS as CS initiates the communication with the IS nodes. CS continues to bear a risk related to its costs (CS 3).

Risks to IS:

IS cluster nodes provide services to IS's primary clients, who are students (and some academics) at UCL. Interactions with cluster nodes should not interfere with the provision of these services. Communicating with CS servers poses the following risks:

IS 4 *The volume of communications may require an unacceptable amount of cluster node resources to process.*

IS 5 *The communications with cluster nodes may be malicious, in that they behave other than as specified in this behavioural description. This is a security risk.*

Network security and utilisation risks also still apply (IS 1, IS 2).

IS is now engaged in direct communication with CS so incurs risk IS 3 relating to the cost of these communications. However IS is now also using computational resources, implying a cost, and therefore the risk that compensation will not be forthcoming.

IS 6 *IS offers computational resources to service requests for processing by CS. This implies a cost, and therefore a risk that IS will not be reimbursed for this cost.*

Risks to the chemist:

This step implies the following risks previously identified: (Chemistry 6, Chemistry 7).

8. The `Condor submit daemon` contacts the allocated grid nodes for a submission, stages the parameters and input files to them and instructs them to begin processing.

   This step implies the same risks as the previous step.

9. The grid nodes process their individual jobs.

   Risks to chemist:

   Delays and faults in the processing of the jobs are risks to the the chemist (Chemistry 6, Chemistry 7).

   Risks to IS:

   IS 7 *The jobs scheduled may place an unacceptable load on the cluster nodes.*

IS 8 *The IS node will be running some software for which IS is not directly responsible. This may cause failures of the node meaning that IS cannot deliver their core services (providing workstations for the university population).*

Also, IS 6.

10. The grid nodes notify the submission node when they have completed their jobs. They then stage the results files back to the submission node.

    This activity involves communication between IS and CS nodes. CS bears a security risk (CS 4), but should otherwise be able to anticipate the volume and size of results. IS also bears a security risk (IS 5). CS and IS both bear cost related risks (CS 3, IS 3, IS 6). The chemist still bears the risks of delays or faults (Chemistry 6, Chemistry 7).

11. The BPEL engine polls the `GridSAM` service for the status of jobs on the queue. The completion of a job may trigger the scheduling of additional jobs. Following `MOLPAK` jobs, `DMAREL` jobs are scheduled. Scheduling of jobs is the responsibility of the CS department.

    This step includes the repetition of steps 5 – 10. All risks associated with this step have been previously identified.

12. Each time a `DMAREL` job completes the workflow engine coordinates the production or update of the results website by invoking the `polyutilsPartner` web-service on the submission node.

    This occurs within the CS network and implies no additional risks.

13. As part of its operation the `polyutilsPartner` web-service invokes the `plotws` web-service in Southampton to prepare a scatter graph of the results. The invocation is synchronous and is between a CS node and a Southampton node. The request and response pass across the CS network, the Internet and the Southampton network. Servicing the request is the responsibility of Southampton University.

    Risks to chemist:

    This step implies the following risks previously identified: (Chemistry 6, Chemistry 7).

    Additionally:

    Chemistry 8 *Results data owned by the chemist will be passed to Southampton via the Internet. The data will be in a form appropriate for plotting, so may not have any intrinsic value. However, if it does, there may be a risk that the data will be intercepted.*

    Risks to CS:

    The cost-related risk CS 3. Also:

    CS 5 *The CS network must be connected to the Internet network and be prepared to accept traffic appearing to originate from certain nodes within Southampton's network. This is a security risk.*

Risks to the ISP:

ISP 1 *The ISPs network must be connected to both the CS network and Southampton's network. This is a security risk.*

ISP 2 *Legitimate traffic between CS and Southampton may (in an extremely improbably worst case) exhaust the ISPs network capacity, hindering their ability to provide network capacity to other clients.*

ISP 3 *The ISP, if it chooses to convey traffic between CS and Southampton, will incur costs. There is the risk that the ISP will not be reimbursed for these costs.*

Risks to Southampton:

Southampton 1 *Southampton's network must be connected to the Internet network and be prepared to accept traffic appearing to originate from certain nodes with UCL CS's network. Southampton's plot server may accept plot requests which may be improperly constructed. This is a security risk.*

Southampton 2 *Southampton must be prepared to process requests for graph plots. Legitimate requests may exhaust Southampton's network or processing resources if they arrive in to great a volume, hindering Southampton in the performance of their usual business.*

Southampton 3 *If Southampton chooses to process requests to the plot service, they will incur costs. There is a risk that Southampton may not be reimbursed for these costs.*

14. The Chemist occasionally checks the results website. When the results are ready, the chemist may view the plot of the results and download result-data files. These interactions occur between a Chemistry and a CS node, and pass across the Chemistry, CS and IS networks.

    Risks to chemist:

    Clearly, the completion or delay of this step represents the ultimate realisation of the cardinal risks to the chemist (Chemistry 6, Chemistry 7), that experiment processing will be completed with serious delays or with errors (causing further delays and/or duplicated effort).

    Risks to CS:

    Security risk CS 2 and cost-related risk CS 3.

    Risks to IS:

    IS 1 and IS 2.

B.2 SLA clauses and risk analysis

B.2.1 SLA 1: Provision of `Polymorph Search Webclient` by IS to Chemistry

Risk mitigation

SLA 1 is the SLA between IS and Chemistry. As the only SLA in the proposed system in which Chemistry participates, SLA 1 is the only SLA with the potential to insure all of the risks to Chemistry that relying on the outsourced grid service implies. For each of the eight risks to Chemistry previously identified, I now either design mitigating conditions to be included in this SLA, or argue that they should be overlooked:

- (Chemistry 1, pg. 265) The risk of delays during experimental setup should be mitigated by **associating penalties with poor performance of the `Polymorph Search Webclient`**, as measured at the interface between the chemistry and IS networks.

  Delays may also be caused by the unavailability of the service. As previously discussed, the monitorable evidence indicating an unavailability may be a sequence of failures, in which case it will be mitigated by the same SLA conditions that will mitigate the risks due to faults in the service. However, the parties **may choose to distinguish unavailability from unreliability by relating penalties to the intervals between bug and bug-fix reports**, thereby allowing differentiation of penalties. This is discussed further below.

- (Chemistry 2, pg. 266) The risk of introducing parameterisation errors during experimental setup should be mitigated in two ways. Firstly by **associating penalties with incorrect behaviours of the `Polymorph Search Webclient`** for intermediate steps in the configuration process (such as when reviewing parameter values). Secondly, by defining any penalties related to the overall correctness of the experimental process in terms of the data provided by the Chemist during configuration requests, as opposed to the possibly incorrect data held by the web-client immediately prior to the commencement of the experiment.

- (Chemistry 3, pg. 266) Concerning the security risk implied by allowing communication with Chemisty nodes by CS nodes: in order to mitigate Chemistry 2, Chemistry 5 and Chemistry 7, which are all risks due to faulty behaviour of the service, the SLA will need to include a more or less formal description of the behaviour of the service, thereby establishing a basis for agreement between the parties on when a fault has occurred. If polymorph search experiments were the only type of interaction between CS nodes and chemistry nodes, other types of interaction with Chemistry's network could be forbidden in the SLA and associated with penalties, termination or breaching of the agreement.

  In fact, Chemistry, IS and CS participate in many other legitimate types of network interactions. All parties in this scenario accept the risks of operating open networks connected to the Internet. Therefore this risk, and security risks identified as applying other parties will not be addressed further in this case-study. This applies to the following risks: Chemistry 3, IS 1, IS 5, CS 1, CS 4, CS 5, ISP 1, and Southampton 1.

Note that the risk of certain types of malicious behaviour associated with network interactions is mitigated by statute. At the time of writing, in the British jurisdiction (in which the parties in this case-study abide) several types of malicious behaviour are legislated against in the criminal law covered by the Computer Misuse Act of 1990.

Other types of malicious behaviour are limited by technical means. The `Polymorph Search Webclient` requires a login to prevent unauthorised access. This mitigates risk CS 1 to some extent. Condor also implements a number of security mechanisms, partially mitigating risks IS 1, IS 5 and CS 4.

- (Chemistry 4, pg. 267) Harm caused by delays occurring during the in the initiation of experiments should be mitigated by **penalties related to the delay taken to acknowledge that the experiment has started**, since this represents time wasted by the chemist waiting to determine whether a fault has occurred.

- (Chemistry 5, pg. 267) **Faults starting the experiment** represent delays and wasted effort for the chemist who must repeatedly attempt to start the experiment. They **should therefore be penalised**.

- (Chemistry 6, pg. 268) The risk of delays in the completion of the experiment, following the triggering of its execution, is the one of the two risks with the potential to cause the most harm to the chemist, the other being Chemistry 7, the risk that a fault will occur during the execution of the experiment. These risks are harmful because experiments take such a long time to complete, meaning that these events may not be detected for a long time, meaning that the delays caused, and the effort required to recover the situation, may be large. Clearly the risk of delays in the overall completion of the experiment must be mitigated by associating **a latency condition over the amount of time it takes for results to become available**.

- (Chemistry 7, pg. 268) Faults may occur during any stage in the processing of the experiment. Many of these faults will be detectable by CS, who will have the opportunity to recover the experiment by repeating all or part of its execution. This will be undetectable by the chemist unless it results in a delay, which will be penalised by a latency condition.

  Some faults may result in **corruption of the results of the experiment**, that CS will miss but the chemist may eventually be able to detect. These **should be penalised by a reliability condition**.

  Catastrophically, the service may fail in such a way that no results of any kind are ever delivered. For this to happen, CS would have to lose the original parameters of the experiment, as may happen in a failure of the submission node. An SLA to penalise this occurrence would rely on evidence of the event happen being communicated between CS and IS, then IS and Chemistry, perhaps using a reporting mechanism similar to that used to terminate an SLA. However, this will be covered by the latency condition, and there will be a cap on the magnitude of penalties for delayed results.

- (Chemistry 8, pg. 270) The risk that information of value to the chemistry department is leaked during the processing of the experiment can be mitigated by associating a penalty with the event

that the chemistry department and IS become aware that a leak has occurred. Assessing the magnitude of this risk, defining it formally and monitoring pertinent events is a difficult problem, probably related to the modelling of other security characteristics, so we defer consideration of this important area to future work.

Risks to IS:

Chemistry's behaviour also results in two previously identified risks to IS. Since SLA 1 is the only agreement made between Chemistry and IS it is the place to address this risks. It is:

- (IS 2, pg. 266) The risk that the volume of service traffic will overwhelm IS's network is realistically a slight one since IS's network will be high capacity compared with anything but the most concerted efforts to produce large volumes of service traffic. Nevertheless, it may be managed by including throughput conditions to limit the amount of experiment configuration, unsuccessful experiment invocation requests, and service result requests that the Chemist may make in an interval of time.

- (IS 3, pg. 267) SLA 1 can be used by IS to require compensation from Chemistry for the provision of the service, hence covering the cost of providing network services. The SLA must therefore include a **payment scheme**.

New risks to Chemistry:

SLA 1, with correctly parameterised clauses as specified above, should mitigate the risks to Chemistry of using the polymorph-search service. However, the decision to enter the SLA poses an additional risk:

Chemistry 9 *Chemistry will probably have to make a small investment to begin using the service, so if SLA 1 is terminated unexpectedly by IS before the end of its agreed period, Chemistry may not get the benefit of this investment. This is a termination risk.*

In the current case-study, we observe that Chemistry is already using the service, and that the development costs invested to do so are nugatory since only a web-browser is required to access the service. However, assuming that Chemistry would have to pay IS to use the service, Chemistry may wish to protect the administrative cost of negotiating the SLA and setting up payment systems. Moreover, Chemistry may want the confidence to make decisions on the basis that the service is available for the foreseeable future.

This risk can only be mitigated in SLA 1, by including a **penalty for termination**.

New risks to IS:

As a result of entering into SLA 1, IS will assume some additional risks due to the behaviour of others that would not otherwise apply:

IS 9 *Delays or faults in the polymorph webclient website, due to the actions of CS, the ISP or Southampton, may cause IS to be liable to pay penalties to Chemistry. This is a safety risk as previous described, in that it is a direct consequence of entering into an SLA.*

This safety risk will be mitigated by SLA 2.

As described in the next section, CS will require a condition in SLA 2 to protect its ability to deliver responses in a timely manner. This results in an additional safety risk for IS.

IS 10 *An excess of requests originating in Chemistry may violate the input throughput condition required by CS in SLA 2, resulting in IS having to pay penalties to CS. This is a safety risk.*

This risk can only be mitigated safely in SLA 1, by imposing a **condition on the rate at which Chemistry can access the polymorph search service**. Such a condition has already been proposed as a remedy for risk IS 2, but I now observe that the parameters of the condition must be compatible with the parameters for CS's condition in SLA 2.

CS will also require a limit on the number of experiments that can be successfully started in a given period of time, in order to protect its ability, and that of the cluster, to calculate results in a timely manner. This results in the following safety risk to IS:

IS 11 *An excess of successfully started experiments originating in Chemistry may violate the condition on successfully started experiments required by CS in SLA 2, resulting in IS having to pay penalties to CS. This is a safety risk.*

This risk can only be mitigated safely in SLA 1, by imposing a **condition on the rate at which Chemistry can successfully start experiments**, compatible with the condition in SLA 2.

As discussed below, CS will require a payment scheme in SLA 2 to cover its costs. This results in the risk:

IS 12 *IS may have to pay CS for the provision of the* `Polymorph Search Webclient` *but may not be paid themselves.*

This is a safety risk, which can be mitigated by increasing the **payment scheme** applied in SLA 1 so that costs implied by the need to pay CS for the service are covered in addition to the cost of providing network services (the cost of providing cluster services is covered in SLA 3).

Chemistry provides IS with the `MOLPAK` and `DMAREL` executables. Chemistry then provides the executables to CS, who provide them back to IS when configuring cluster nodes. IS ultimately guarantees the performance of individual cluster jobs in SLA 3, meaning that they need a guarantee of the performance of `MOLPAK` and `DMAREL` from CS. Therefore, CS needs a guarantee of the performance from IS in SLA 2 (note that this is not quite purely reciprocal, because CS could in theory modify the executables before configuring the nodes). Therefore, finally, there may be a safety risk that IS is obliged to guarantee the performance of `MOLPAK` and `DMAREL` to CS without receiving equivalent guarantees from Chemistry.

IS 13 *IS must guarantee the performance of the* `MOLPAK` *and* `DMAREL` *executables to CS in SLA 2. This represents a safety risk if IS cannot obtain equivalent guarantees from Chemistry.*

This risk can only be addressed by **obtaining a guarantee concerning the performance of `MOLPAK` and `DMARE` from Chemistry**

Finally, by entering into SLA 1, IS also assumes a termination risk.

IS 14 *IS will probably have to make a small investment to begin using the service, so if SLA 1 is terminated unexpectedly by Chemistry before the end of its agreed period, IS may not get the benefit of this investment. This is a termination risk.*

This risk can only be mitigated in SLA 1, by including a **penalty for termination**.

Summary of conditions

1. Latency conditions on the setup operations of the webclient;
2. optionally, availability conditions relating to the service;
3. reliability conditions on the setup operation of the webclient;
4. latency condition on the experiment invocation operation of the webclient;
5. reliability condition on the experiment invocation operation of the webclient;
6. latency condition on the amount of time taken for experimental results to become available;
7. reliability condition on results retrieval operations;
8. throughput conditions on all operations;
9. a payment scheme, charging Chemistry for use of the service;
10. a termination penalty for IS terminating the service;
11. a limit on the rate at which experiments can be started;
12. a guarantee concerning the performance of `MOLPAK` and `DMAREL` from Chemistry;
13. a termination penalty for Chemistry terminating the service.

Service interface

`http://sse.cs.ucl.ac.uk/omii-bpel/polymorph/index.htm` GET Returns main frame with polymorph-search parameter-file frame embedded, and links to invoke frame and results frame.
`http://trout1.cs.ucl.ac.uk:18080/PolymorphSearchWebClient/-PolymorphSearch.htm` GET Form for specifying files to upload for Fileuploader.jsp
`http://trout1.cs.ucl.ac.uk:18080/PolymorphSearchWebClient/-Invoke.htm` GET Form for specifying parameters for Invoke.jsp

Table B.1: HTTP service interface to the `Polymorph search webclient`, returning static pages

The interface to the `Polymorph Search Webclient` is accessed using the HTTP protocol. Requests to the webclient fall into three main categories; several pages present user interface components

`http://trout1.cs.ucl.ac.uk:18080/PolymorphSearchWebClient/-Fileuploader.jsp` POST Parameters are: `fileBondlengths` – a file used to determine which are the covalent bonds within the structure, defining the organic molecule to be held rigid `fileCadpacCharges` – a file describing the charge density of the molecules in terms of charges, dipoles, quadrupoles, etc. at each atom `fileDmarelAxis` – a file defining the axis system needed to define dipoles, quadrupoles, etc. in cadpac.charges. `fileMolpakXyz` – a file giving the coordinates and atom types of the molecule, the only input needed for MOLPAK `filePoteDat` – a file defining the parameters for the model for the repulsion and dispersion forces between the molecules Expected results: A page acknowledging the upload of the parameter files
`http://trout1.cs.ucl.ac.uk:18080/PolymorphSearchWebClient/-Invoke.jsp` POST parameters are: `analysisID18` - a string identifying the experiment `pt` – a list of packing types to consider Expected results: A page acknowledging the successful invocation of an experiment

Table B.2: HTTP service interface to the `Polymorph search webclient`, for submission of configuration files and execution of experiments

for the convenience of specifying parameters for requests that have an effect configuring an experiment or launching it; other pages accept these parameters and cause a change in the state of the service; finally, a set of pages provides access to the results of the service.

URLs, access methods, and intended results for the first category of pages, static configuration pages, are listed in Table B.1.

The second category of pages, parameter submission and experiment execution pages, are listed in Table B.2.

The third category of pages, results, are listed in Table B.3. The URLs for results pages are specific to particular experiments. `ID` in the URLs listed represents the identifier string specified as a parameter for the experiment:

B.2.2 SLA 2: Provision of `Polymorph Search Webclient` by CS to IS

Risk mitigation

The following risks to IS should be addressed by an SLA with CS in respect of the `Polymorph Search Webclient`:

- (IS 9, pg. 9) If delays and faults occur in the web-client or during the processing of an experiment, IS will be liable to pay penalties to Chemistry. The faults will be monitorable on the interface between IS and CS. Hence, **for all clauses identified in SLA 1 that may result in IS paying a penalty to CS, IS should demand a compatible clause in SLA 2 from CS**. The parameters for the guarantees offered by IS to Chemistry will be looser than those offered by CS to IS to accommodate extra faults and delays introduced by the IS network.

http://trout1.cs.ucl.ac.uk:18080/axis/polymorph GET Listing of results pages
`http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/ID.html` GET A page summarising the completed DMAREL executions
`http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/ID.png` GET A scatter graph summarising the completed DMAREL executions
`http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/ID.xml` GET Experimental results in XML format
`http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/ID/-bondlengths` GET The original bondlengths file, uploaded before the experiment was invoked
`http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/ID/-cadpac.charges` GET The original cadpak.charges file, uploaded before the experiment was invoked
`http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/ID/dmarel.axis` GET The original dmarel.axis file, uploaded before the experiment was invoked
`http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/ID/molpak.xyz` GET The original molpak.xyz file, uploaded before the experiment was invoked
`http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/ID/pote.dat` GET The original pote.dat file, uploaded before the experiment was invoked

Table B.3: HTTP service interface to results generated by the `Polymorph search webclient`

In addition, CS will wish to mitigate the following risks:

- (CS 2, pg. 267) Too much request throughput could diminish the efficiency of the web-client. In addition the uploading of too much data could overload the resources of the server on which the web-client resides. These risks can be mitigated by assigning input-throughput constraints to the relevant configuration operations. The functional behaviour of the service will guarantee that disk resources are not violated, since the size of configuration files is bounded, and an input throughput constraint will limit the number of unique configurations possible.

- (CS 2, pg. 267) CS can mitigate the risk of not being paid for providing the polymorph-search service by **requiring payment from IS** in SLA 2.

- (CS 10, pg. 282) In SLA 3, IS requires a guarantee from CS concerning the performance of the versions of `MOLPAK` and `DMAREL` used in cluster jobs. This represents a safety risk to CS, therefore it must obtain **a guarantee concerning the performance of the versions of `MOLPAK` and `DMAREL` provided by IS**.

New risks to IS:

By entering into SLA 2, IS will solve the problem of their safety risk, but at the expense of a reliance on SLA 2, which is a termination risk:

IS 15 *IS relies on SLA 2 to mitigate the safety risks implied by SLA 1. If SLA 2 terminates, then IS will probably need to terminate SLA 1 and therefore pay a penalty. This is a termination risk.*

This risk can only be mitigated by including a **termination penalty** in SLA 2.

New risks to CS:

CS 6 *By guaranteeing latency and reliability properties of the overall experiment service to IS, CS assumes the additional risk that the experiment will be faulty or delayed due to problems occurring in IS cluster machines, the Internet or Southampton's plotting service. This is a safety risk.*

This risk is mitigated in SLAs 3 and 4.

CS 7 *The cluster has a finite capacity, access to a proportion of which is insured by SLA 3. Over-utilisation of the service may exhaust the capacity of the cluster, causing CS to be incapable of meeting latency constraints in SLA 2.*

This risk must be addressed by a **throughput constraint on successfully started experiments**.

If CS is going to offer guarantees on the latency of experimental result production, it will need to obtain guarantees concerning the behaviour of `MOLPAK` and `DMAREL` from IS, so it can offer these guarantees back to IS in SLA 3.

CS 8 *CS must provide guarantees to IS concerning the behaviour of `MOLPAK` and `DMAREL` in SLA 3. This represents a risk if CS does not receive a compatible guarantee from IS.*

IS must provide a **guarantee concerning the performance of `MOLPAK` and `DMAREL`** to CS in SLA 2.

CS may perceive a risk due to loss of income from the polymorph-search service if this SLA were terminated:

CS 9 *The termination of SLA 2 would potentially deprive CS of income from the polymorph-search service, and may leave CS with residual payment responsibilities provided by IS or the ISP in SLAs 3 and 4. This is a termination risk.*

This risk may be mitigated by a **termination penalty** included in SLA 2.

Summary of conditions

1. Latency conditions on the setup operations of the webclient;
2. optionally, availability conditions relating to the service;
3. reliability conditions on the setup operation of the webclient;
4. latency condition on the experiment invocation operation of the webclient;
5. reliability condition on the experiment invocation operation of the webclient;
6. latency condition on the amount of time taken for experimental results to become available;
7. reliability condition on results retrieval operations;
8. throughput conditions on all operations;
9. a payment scheme, charging IS for use of the service;
10. a guarantee concerning the performance of the versions of `MOLPAK` and `DMAREL` provided by IS to CS;
11. a termination penalty for CS terminating the service;
12. a limit on the rate at which experiments can be started;
13. a termination penalty for IS terminating the service.

Service interface

The service interface for SLA 2 is the same as that for SLA 1.

B.2.3 SLA 3: Provision of `Condor` cluster services by IS to CS

Risk mitigation

SLA 3 covers the interaction between the submission and Condor manager nodes in the CS network, and cluster nodes provided by IS.

Risks to CS:

- (IS 6, pg. 279) the safety risk that CS assumes when it offers guarantees concerning the overall experimental execution time, specifically the part of that risk due to the potential for poor performance from the cluster nodes.

Determining how CS's safety risk should properly be mitigated is complicated by the fact that the precise protocols by which `Condor` nodes communicate with the `Condor controller` and the `Condor submit daemon` are not documented and reverse engineering `Condor` represents a greater effort than can be afforded in this case study. However, I speculate that the SLA should provide guarantees as follows:

Cluster nodes provide notifications to the `Condor Manager` as to their availability for processing tasks. The SLA should specify that **at any given time, a certain number of nodes should be apparently either available for processing or already assigned to a job**. The SLA should also specify that **with a high reliability, nodes advertising themselves as available for processing will also accept a job assignment**. This will insure that CS has access to a predictable amount of processing capacity. **Jobs should then be subject to latency and reliability constraints** to ensure that they are genuinely getting their fair share of node resources.

Note that in this respect SLA 3 represents a gamble for IS. If all of their machines are suddenly required by students, they will be required to pay CS for the lack of availability. However, IS is the only party with the power to control this risk, for example by restricting student access to cluster rooms, or installing new cluster nodes to improve capacity. It must therefore be prevailed up to mitigate the risk.

Clearly in order to execute this job assignment protocol reliably, **latency and reliability constraints will have to apply at a fine granularity to the exchange of messages between IS and CS**.

Risks to IS:

- (IS 4, pg. 269) The risk that legitimate communications with the cluster nodes may require an unacceptable amount of cluster node resources may be mitigated by a **throughput condition on cluster node operation invocations**.

- (IS 6, pg. 269) The cost-related risk that IS assumes providing cluster services to CS must be mitigated by incorporating aa **payment scheme** into SLA 3.

- (IS 7,, pg. 269) The risk that the load placed on cluster nodes is too great must be mitigated by imposing **a limit on job assignments to cluster nodes**. Whether this would be a rate limitation or related to the configuration of cluster nodes is not currently known.

- (IS 8, pg. 270) The risk that IS assumes by running software provided by a third party, is hard to mitigate in an SLA, so will be ruled out. In practice, this type of risk is better mitigated either through some kind of certification scheme, or by running software in a secure manner so that it cannot interfere with the node on which it is executing.

New risks to IS:

IS will essentially be providing a delegated execution service in SLA 3. The performance of such a service will depend on the performance of the executables provided to it, in this case `MOLPAK` and `DMAREL`. Therefore, to offer latency guarantees for the overall completion of a job, IS will require a guarantee in respect of the performance of these executables from CS.

IS 16 *The overall completion time of a cluster job depends in part on the executable being executed. If IS guarantees this latency without receiving a guarantee on the performance of the executable, this represents a safety risk.*

This risk can only be mitigated in SLA 3 by **requiring guarantees from CS regarding the execution time of `MOLPAK` and `DMAREL`**.

New risks to CS:

CS must provide IS with a guarantee of the performance of `MOLPAK` and `DMAREL`. However, these executables will be provided by IS to CS originally. If CS does not obtain a guarantee of performance of the executables from IS, then it cannot offer the guarantee back.

CS 10 *CS must make a guarantee concerning the performance of `MOLPAK` and `DMAREL` to IS. If it does not obtain a compatible guarantee from IS, this represents a safety risk.*

This safety risk is mitigated in SLA 2.

Summary of constraints

1. A throughput condition on cluster node operation invocations;
2. a condition relating to the number of nodes available for job assignment, or running jobs;
3. a reliability condition on starting new jobs on nodes;
4. a latency condition on job execution;
5. a reliability condition on job execution;
6. latency conditions on communications with the nodes;
7. reliability conditions on communications with the nodes;
8. a payment scheme;
9. a limit on job assignments to nodes;
10. a guarantee concerning the execution times of `MOLPAK` and `DMAREL` made by CS.

Service interface

At present the prohibitive degree of effort required to reverse engineer `Condor` means that the service interface to `Condor` nodes remains obscure.

B.2.4 SLA 4: Provision of `plotws` web-service by ISP to CS

Risk mitigation

This SLA addresses the remainder of the safety risk to CS of guaranteeing a latency constraint on the availability of experimental results. These results include a graph plotted by the `plotws` web-service provided by Southampton university. Delays or errors producing this graph will result in CS being liable to pay penalties to IS. In order that guarantees provided to CS be monitorable, the service must be resold by a network service provider who can provide a connection between CS and Southampton (or more

generally a chain of such parties with a chain of agreements; however, we assume the existence of a single party).

Risks to CS:

- (CS 6) the risk that the guarantees offered by CS in terms of latency and reliability for the completion of experiments will be impossible to meet due to faults or delays in the plotting service should be addressed by **reliability, latency and possibly availability constraints placed on the plotting service** as delivered at the interface between the networks owned by CS and the ISP.

Risks to ISP:

- (ISP 2, pg. 271) The risk that legitimate traffic will overwhelm the ISPs network capacity is of course largely irrelevant to the production of these SLAs, since the ISP will no doubt have vastly greater capacity than could be exhausted by anything other than a concerted malicious effort, which SLAs could do little to prevent. However, **an input-throughput constraint on operation requests** will limit the volume of traffic, and more importantly also address the ISPs safety risk (ISP 5, pg. 284) due to SLA 5.

- (ISP 3, pg. 271) The risk that the ISP will not be able to cover the cost of providing network services to CS and Southampton may be mitigated by including a **payment scheme** in SLA 4. This also covers a safety risk (ISP 6, pg. 285) implied by Southampton's requirement for payment in SLA 5.

New risks to CS:

CS may wish to protect their investment in integrating the plot service (even though they did so on faith in the first instance), mitigating a termination risk:

CS 11 *SLA 4 may represent a termination risk to CS, because the unavailability of the plot service would force CS to find or implement another similar service at short notice.*

This risk can be mitigated in SLA 4 with a **termination penalty**.

The ISP, as a reseller of the service, assumes the following additional risk:

ISP 4 *The capability of the ISP to deliver the plotting service in a reliable and timely manner to CS depends on it being delivered by Southampton in a reliable and timely manner to the ISP. If this does not happen the ISP may be liable to pay penalties to CS.*

This is a safety risk that is addressed in SLA 5. An additional safety risk is a consequence of the need to enter SLA 5. Southampton will require an input-throughput clause to protect their ability to deliver timely and correct results. The ISP must have an equivalent constraint, or else be liable to pay penalties to Southampton if CS exceeds Southampton's permitted capacity.

The ISP doesn't seem to suffer from major termination risk in this case, but may wish to safeguard their income with a **termination penalty**.

Summary of conditions

1. A latency condition on plot operations;
2. a reliability condition on plot operations;
3. an availability condition on plot operations;
4. a payment scheme;
5. a termination penalty for the ISP cancelling the SLA;
6. an input-throughput constraint on operations;
7. a termination penalty for CS cancelling the SLA.

Service interface

The `plotws` service is a web-service specified at `http://plotws.omii.ac.uk:18080/PlotWS/services/Graph?wsdl`. It is accessed using SOAP over HTTP, and provides five operations as follows.

The precise graphical presentation and output format (SVG or PNG) of the graph is controlled by the `opt` parameter, which is of a complex schema type.

The current implementation of the polymorph-search service only uses the `makeplot_xy` operation.

B.2.5 SLA 5: Provision of Plot service by Southampton to IS

Risk mitigation

This SLA addresses the safety risk to the ISP of insuring the performance of Southampton's plotting service.

- (ISP 4) The risk that the ISP will have to pay penalties to CS for the poor performance of the `plotws` service should be mitigated by including in SLA 5 clauses compatible with the **latency, reliability and availability conditions** included in SLA 5.

This SLA also mitigates two risks previously identified to Southampton:

- (Southampton 2) The risk that Southampton's server will be overwhelmed by plot requests should be mitigated by an **input-throughput condition** included in SLA 5.
- (Southampton 3) The risk that Southampton will not receive compensation for their service should be mitigated by including a **payment scheme** in SLA 5.

New risks to the ISP:

The inclusion of an input-throughput condition in SLA 5 implies a safety risk for the ISP, which is mitigated in SLA 4

ISP 5 *There is a danger that requests made by CS will exceed the input-throughput condition in SLA 5, causing the ISP to be liable to pay penalties to Southampton.*

`makePlot_xyy` – plot a graph with one X-series and multiple Y-series In parameters: `xi` – X-series data `yi` – Y-series data (array of arrays of doubles) `opt` – Plot options Out parameters: `makePlotReturn` – The plotted graph
`makePlot_xy` – plot a graph with an X and Y series In parameters: `xi` – X-series data `yi` – Y-series data `opt` – Plot options Out parameters: `makePlotReturn` – The plotted graph
`makePlot_x` – plot a graph with an X series only In parameters: `xi` – X-series data `opt` – Plot options Out parameters: `makePlotReturn` – The plotted graph
`makePlot_xxyy` – plot a graph with multiple X and Y series In parameters: `xi` – X-series data (array of arrays) `yi` – Y-series data (array of arrays) `opt` – Plot options Out parameters: `makePlotReturn` – The plotted graph
`makePlot3d_xyz` – Plot a surface In parameters: `xi` – X-series data `yi` – Y-series data `zi` – Z-series data `opt` – Plot options Out parameters: `makePlotReturn` – The plotted graph

Table B.4: SOAP interface to the `plotws` webservice

If the ISP is obliged to pay Southampton for the use of the service, there is a risk that the ISP may not be able to cover the cost of the service.

ISP 6 *The ISP must pay Southampton for the use of the* `plotws` *service. This is a risk if the ISP does not obtain compensation to cover this cost.*

This risk is mitigated in SLA 4.

The ISP enters into SLA 5 in order to mitigate its safety risk from entering SLA 4. Therefore SLA 5 represents a termination risk to the ISP:

ISP 7 *If Southampton terminates SLA 5, the ISP will need to find a replacement service or terminate SLA 4. Either course will financially disadvantage the ISP, particularly since SLA 4 includes a termination penalty clause. SLA 5 is therefore a termination risk for the ISP.*

This risk should be mitigated by a **termination penalty** in SLA 5.

The plot service is a fairly simple and generic service. Southampton is therefore unlikely to be unduely concerned about termination, as income associated with SLA 5 would likely be small and other customers potentially available. However, they may also desire a **termination penalty**.

There are no new risks for Southampton implied by entering into SLA 5.

Summary of conditions

1. Latency condition on plot operations;
2. reliability condition on plot operations;
3. availability condition on plot operations;
4. a termination penalty for the Southampton cancelling the SLA;
5. an input-throughput constraint on operations;
6. a termination penalty for the ISP cancelling the SLA.

Service interface

The service interface is the same as for SLA 4.

B.3 Case-study risks by party

B.3.1 Chemistry

- (Chemistry 1, pg. 265) Delays in configuring the experiment could significantly delay access by the chemist to the results of the experiment. This could mean missed deadlines, an inability to produce timely results in comparison to researchers from other universities, and similar material harm to the individual chemist and the Chemistry Department.

  Mitigated by: SLA 1, pg. 272

- (Chemistry 2, pg. 266) Faults may occur at any point in the infrastructure supporting the process. Obvious faults may be temporary, or require intervention by another stakeholder.

  Non-obvious faults may result in the corruption of the parameters or the input files for the experiment. In theory these faults could occur at any point in the infrastructure. Non-obvious faults become increasingly problematic while they remain undetected. The worst-case scenario in this case is that corruption of data results in the production of invalid scientific results that are relied upon in future by the Chemists.

  Mitigated by: SLA 2, pg. 272

- (Chemistry 3, pg. 266) The chemistry network must be linked to the IS network and communicate with the web-client. It must therefore be prepared to accept network traffic from the IS network that appears to originate from the CS network. The traffic may potentially act in a manner inconsistent with this behavioural description, either maliciously or accidentally, causing faults or resource exhaustions that reduce the capacity of the Chemistry nodes to perform their usual duties. This is a security risk.

  Mitigated by: Ruled out, pg. 272

- (Chemistry 4, pg. 267) Delays starting the experiment are a risk to the Chemist because the Chemist wastes time waiting for an acknowledgement, and the production of results is also delayed.

  Mitigated by: SLA 1, pg. 273

- (Chemistry 5, pg. 267) Faults starting the experiment are a risk to the Chemist because the Chemist may wastes time in the attempt.

  Mitigated by: SLA 1, pg. 273

- (Chemistry 6, pg. 268) All steps must be completed in a timely fashion for the experiment to be achieved in a timely fashion.

  Mitigated by: SLA 1, pg. 273

- (Chemistry 7, pg. 268) All steps must be completed in such a manner as to generate correct results. Hence, data must not be corrupted during transfer, and all components must behave correctly according to this behavioural specification.

  Mitigated by: SLA 1, pg. 273

- (Chemistry 8, pg. 270) Results data owned by the chemist will be passed to Southampton via the Internet. The data will be in a form appropriate for plotting, so may not have any intrinsic value. However, if it does, there may be a risk that the data will be intercepted.

  Mitigated by: Ruled out, pg. 273

- (Chemistry 9, pg. 274) Chemistry will probably have to make a small investment to begin using the service, so if SLA 1 is terminated unexpectedly by IS before the end of its agreed period, Chemistry may not get the benefit of this investment. This is a termination risk.

  Mitigated by: SLA 1, pg. 274

B.3.2 IS

- (IS 1, pg. 266) Traffic appearing to originate from nodes within the Chemistry and CS networks may be problematic if it acts maliciously. This is a security risk.

  Mitigated by: Ruled out, pg. 272

- (IS 2, pg. 266) The IS network must be connected to the Chemistry and CS networks. It must be prepared to accept legitimate traffic from both networks appearing to originate from nodes within those networks. This traffic may be problematic if it appears in too great a volume.

  Mitigated by: SLA 1, pg. 274

- (IS 3, pg. 267) Conveying legitimate traffic between Chemistry and CS implies a cost to IS. This represents a risk to IS because they may not receive compensation for providing the service.

  Mitigated by: SLA 1, pg. 274

- (IS 4, pg. 269) The volume of communications may require an unacceptable amount of cluster node resources to process.

  Mitigated by: SLA 3, pg. 281

- (IS 5, pg. 269) The communications with cluster nodes may be malicious, in that they behave other than as specified in this behavioural description. This is a security risk.

  Mitigated by: Ruled out, pg. 272

- (IS 6, pg. 269) IS offers computational resources to service requests for processing by CS. This implies a cost, and therefore a risk that IS will not be reimbursed for this cost.

  Mitigated by: SLA 3, pg. 281

- (IS 7, pg. 269) The jobs scheduled may place an unacceptable load on the cluster nodes.

  Mitigated by: SLA 3, pg. 281

- (IS 8, pg. 270) The IS node will be running some software for which IS is not directly responsible. This may cause failures of the node meaning that IS cannot deliver their core services (providing workstations for the university population).

  Mitigated by: Ruled out, pg. 281

- (IS 9, pg. 274) Delays or faults in the polymorph webclient website, due to the actions of CS, the ISP or Southampton, may cause IS to be liable to pay penalties to Chemistry. This is a safety risk as previous described, in that it is a direct consequence of entering into an SLA.

  Mitigated by: SLA 2, pg. 277

- (IS 10, pg. 275) An excess of requests originating in Chemistry may violate the input throughput constraint required by CS in SLA 2, resulting in IS having to pay penalties to CS. This is a safety risk.

  Mitigated by: SLA 1, pg. 275

- (IS 11, pg. 275) An excess of successfully started experiments originating in Chemistry may violate the constraint on successfully started experiments required by CS in SLA 2, resulting in IS having to pay penalties to CS. This is a safety risk.

  Mitigated by: SLA 1, pg. 275

- (IS 12, pg. 275) IS may have to pay CS for the provision of the `Polymorph Search Webclient` but may not be paid themselves.

  Mitigated by: SLA 1, pg. 274

- (IS 13, pg. 275) IS must guarantee the performance of the `MOLPAK` and `DMAREL` executables to CS in SLA 2. This represents a safety risk if IS cannot obtain equivalent guarantees from Chemistry.

  Mitigated by: SLA 1, pg. 275

- (IS 14, pg. 276) IS will probably have to make a small investment to begin using the service, so if SLA 1 is terminated unexpectedly by Chemistry before the end of its agreed period, IS may not get the benefit of this investment. This is a termination risk.

  Mitigated by: SLA 1, pg. 276

- (IS 15, pg. 279) IS relies on SLA 2 to mitigate the safety risks implied by SLA 1. If SLA 2 terminates, then IS will probably need to terminate SLA 1 and therefore pay a penalty. This is a termination risk.

  Mitigated by: SLA 2, pg. 279

- (IS 16, pg. 282) The overall completion time of a cluster job depends in part on the executable being executed. If IS guarantees this latency without receiving a guarantee on the performance of the executable, this represents a safety risk.

  Mitigated by: SLA 3, pg. 282

B.3.3 CS

- (CS 1, pg. 267) The CS network must be connected to the IS network. It must be prepared to receive traffic that appears to have originated from nodes in the chemistry department. This may present a security risk.

  Mitigated by: Ruled out, pg. 272

- (CS 2, pg. 267) Legitimate behaviour may also be problematic if it arrives in too great a volume leading to resource exhaustion. CS servers are a valuable resource, and may not be used exclusively to provide the polymorph search service. Therefore it would be senseless for CS to allow its submission node to be overwhelmed by requests. Webpage requests may reduce the functionality of the `Polymorph Search Webclient` if they arrive in too great a volume. The upload of large amounts of data may exhaust the storage capacity of a CS server, rendering it useless for other purposes and clients.

  Mitigated by: SLA 2, pg. 279

- (CS 3, pg. 267) Servicing operation requests submitted to the `Polymorph Search Webclient` implies costs for CS due to the provision of network and processing resources. This implies a risk to CS that they will not receive adequate compensation to cover these costs.

  Mitigated by: SLA 2, pg. 279

- (CS 4, pg. 269) CS servers must communicate with nodes managed by IS. This may constitute a security risk.

  Mitigated by: Ruled out, pg. 272

- (CS 5, pg. 270) The CS network must be connected to the Internet network and be prepared to accept traffic appearing to originate from certain nodes within Southampton's network. This is a security risk.

Mitigated by: Ruled out, pg. 272

- (CS 6, pg. 279) By guaranteeing latency and reliability properties of the overall experiment service to IS, CS assumes the additional risk that the experiment will be faulty or delayed due to problems occurring in IS cluster machines, the internet or Southampton's plotting service. This is a safety risk.

  Mitigated by: SLAs 3 and 4, pgs. 280, 283

- (CS 7, pg. 279) The cluster has a finite capacity, access to a proportion of which is insured by SLA 3. Over-utilisation of the service may exhaust the capacity of the cluster, causing CS to be incapable of meeting latency constraints in SLA 2.

  Mitigated by: SLA 2, pg. 279

- (CS 9, pg. 280) The termination of SLA 2 would potentially deprive CS of income from the polymorph-search service. This is a termination risk.

  Mitigated by: SLA 2, pg. 280

- (CS 10, pg. 282) CS must make a guarantee concerning the performance of `MOLPAK` and `DMAREL` to IS. If it does not obtain a compatible guarantee from IS, this represents a safety risk.

  Mitigated by: SLA 2, pg. 279

- (CS 11, pg. 283) SLA 4 may represent a termination risk to CS, because the unavailability of the plot service would force CS to find or implement another similar service at short notice.

  Mitigated by: SLA 4, pg. 283

B.3.4 ISP

- (ISP 1, pg. 271) The ISPs network must be connected to both the CS network and Southampton's network. This is a security risk.

  Mitigated by: Ruled out, pg. 272

- (ISP 2, pg. 271) Legitimate traffic between CS and Southampton may (in an extremely improbably worst case) exhaust the ISPs network capacity, hindering their ability to provide network capacity to other clients.

  Mitigated by: Ruled out, pg. 283

- (ISP 3, pg. 271) The ISP, if it chooses to convey traffic between CS and Southampton, will incur costs. There is the risk that the ISP will not be reimbursed for these costs.

  Mitigated by: SLA 4, pg. 283

- (ISP 4, pg. 283) The capability of the ISP to deliver the plotting service in a reliable and timely manner to CS depends on it being delivered by Southampton in a reliable and timely manner to the ISP. If this does not happen the ISP may be liable to pay penalties to CS.

  Mitigated by: SLA 5, pg. 284

- (ISP 5, pg. 284) There is a danger that requests made by CS will exceed the input-throughput condition in SLA 5, causing the ISP to be liable to pay penalties to Southampton.

- (ISP 6, pg. 285) The ISP must pay Southampton for the use of the `plotws` service. This is a risk if the ISP does not obtain compensation to cover this cost.

  Mitigated by: SLA 4, pg. 283

- (ISP 7, pg. 285) If Southampton terminates SLA 5, the ISP will need to find a replacement service or terminate SLA 4. Either course will financially disadvantage the ISP, particularly since SLA 4 includes a termination penalty clause. SLA 5 is therefore a termination risk for the ISP.

  Mitigated by: SLA 5, pg. 285

B.3.5 Southampton

- (Southampton 1, pg. 271) Southampton's network must be connected to the Internet network and be prepared to accept traffic appearing to originate from certain nodes with UCL CS's network. Southampton's plot server may accept plot requests which may be improperly constructed. This is a security risk.

  Mitigated by: Ruled out, pg. 272

- (Southampton 2, pg. 271) Southampton must be prepared to process requests for graph plots. Legitimate requests may be harmful if they arrive in to great a volume.

  Mitigated by: SLA 5, pg. 284

- (Southampton 3, pg. 271) If Southampton chooses to process requests to the plot service, they will incur costs. There is a risk that Southampton may not be reimbursed for these costs.

  Mitigated by: SLA 5, pg. 284

Appendix C

SLA 1: Chemistry and IS

```
/*
This is an SLA written using the Human−Usable Textual Notation (HUTN) for
the language SLAng.

HUTN is a concrete−syntax standard of the OMG (available http://www.omg.org/)

The SLA is the only object with type ::slang::MutuallyMonitorableSLA.
*/

specification =
    "http://uclslang.sourceforge.net/specifications/thesis−combined.emofxmi"

configuration =
    "http://uclslang.sourceforge.net/specifications/thesis−combined.hutn"

// SLA starts here

using ::slang {

    MutuallyMonitorableSLA("SLA between Chemistry and IS") {

        uRI = "http://uclslang.sourceforge.net/thesis/sla1.hutn";

        parties = {

            PartyDefinition[chemistry](
            "The Department of Chemistry, University College London"),

            PartyDefinition[uclis](
            "Information Services, University College London")
        }

        services = {

            es::ElectronicServiceDefinition[polymorph](
            "The provision of the Polymorph Search Webclient by IS to Chemistry"
            ) {

                provider = PartyDefinition[uclis]

                client = PartyDefinition[chemistry]

                interfaces = {
```

```
45  es::ElectronicServiceInterfaceDefinition[polymorph](
46      "HTTP interface to the Polymorph Search Webclient") {

48      owner = PartyDefinition[uclis]

50      operations = {

52          es::OperationDefinition[static1](
53              "http://sse.cs.ucl.ac.uk/omii-bpel/polymorph/index.htm") {

55              parameters = {

57                  es::ParameterDefinition(
58                      "HTTP response status code", OUT),
59                  es::ParameterDefinition(
60                      "HTTP response message body", OUT)
61              }
62          },
63          es::OperationDefinition[static2](
64              "http://trout1.cs.ucl.ac.uk:18080/PolymorphSearchWebClient/—
                    PolymorphSearch.htm") {

66              parameters = {

68                  es::ParameterDefinition(
69                      "HTTP response status code", OUT),
70                  es::ParameterDefinition(
71                      "HTTP response message body", OUT)
72              }
73          },
74          es::OperationDefinition[static3](
75              "http://trout1.cs.ucl.ac.uk:18080/PolymorphSearchWebClient/—
                    Invoke.htm") {

77              parameters = {

79                  es::ParameterDefinition(
80                      "HTTP response status code", OUT),
81                  es::ParameterDefinition(
82                      "HTTP response message body", OUT)
83              }
84          },
85          es::OperationDefinition[submit](
86              "http://trout1.cs.ucl.ac.uk:18080/PolymorphSearchWebClient/—
                    Fileuploader.jsp") {

88              parameters = {

90                  es::ParameterDefinition("Post parameter fileBondlengths - —
                        determining the covalent bonds within the structure, —
                        defining the organic molecule to be held rigid", IN),
91                  es::ParameterDefinition("Post parameter fileCadpacCharges —
                        - describing the charge density of the molecules in —
                        terms of charges, dipoles, quadrupoles, etc. at each —
                        atom", IN),
92                  es::ParameterDefinition("Post parameter fileDmarelAxis - —
                        a file defining the axis system needed to define dipoles,—
```

```
                quadrupoles, etc. in cadpac.charges", IN),
93              es::ParameterDefinition("Post parameter fileMolpakXyz - a—
                    file giving the coordinates and atom types of the —
                    molecule", IN)
94              es::ParameterDefinition("Post parameter filePoteDat - a file—
                    defining the parameters for the model for the repulsion—
                    and dispersion forces between the molecules", IN),
95              es::ParameterDefinition("HTTP response status code", OUT—
                    ),
96              es::ParameterDefinition("HTTP response message body", —
                    OUT)
97          }
98      },
99      ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke](
100         "Operation: makePlot_xyy") {

102         parameters = {

104             es::ParameterDefinition[invokeId]("Post parameter —
                    analysisID18 - the ID of the experiment", IN),
105             es::ParameterDefinition("Post parameter pt - a list of —
                    packing types to consider", IN),
106             es::ParameterDefinition("HTTP response status code", OUT—
                    ),
107             es::ParameterDefinition("HTTP response message body", —
                    OUT)
108         }

110         executables = {

112             ::sla1::slang::es::FixedDurationExecutableDefinition[—
                    molpak](
113                 "The MOLPAK executable") {

115                 referenceNodeSpeed = 1.0;

117                 maxDuration = ::types::Duration(1, hr)

119                 maintainer = PartyDefinition[chemistry]
120             },

122             ::sla1::slang::es::FixedDurationExecutableDefinition[dmarel—
                    ](
123                 "The DMAREL executable") {

125                 referenceNodeSpeed = 1.0;

127                 maxDuration = ::types::Duration(1, hr)

129                 maintainer = PartyDefinition[chemistry]
130             }
131         }
132     },

134     es::OperationDefinition[results](
135         "Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph") —
                {
```

```
137                 parameters = {

139                     es::ParameterDefinition(
140                         ”HTTP response status code”, OUT),
141                     es::ParameterDefinition[resultsId](
142                         ”HTTP response message body”, OUT)
143                 }
144             },

146             es::OperationDefinition[results1](
147                 ”Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/—
                        ID.html”) {

149                 parameters = {

151                     es::ParameterDefinition[results1Id](
152                         ”Experiment ID, a component of the operation URL”, —
                                IN),
153                     es::ParameterDefinition(
154                         ”HTTP response status code”, OUT),
155                     es::ParameterDefinition(
156                         ”HTTP response message body”, OUT)
157                 }
158             },
159             es::OperationDefinition[results2](
160                 ”Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/—
                        ID.png”) {

162                 parameters = {

164                     es::ParameterDefinition[results2Id](
165                         ”Experiment ID, a component of the operation URL”, —
                                IN),
166                     es::ParameterDefinition(
167                         ”HTTP response status code”, OUT),
168                     es::ParameterDefinition(
169                         ”HTTP response message body”, OUT)
170                 }
171             },
172             es::OperationDefinition[results3](
173                 ”Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/—
                        ID.xml”) {

175                 parameters = {

177                     es::ParameterDefinition[results3Id](
178                         ”Experiment ID, a component of the operation URL”, —
                                IN),
179                     es::ParameterDefinition(
180                         ”HTTP response status code”, OUT),
181                     es::ParameterDefinition(
182                         ”HTTP response message body”, OUT)
183                 }
184             },
185             es::OperationDefinition[results4](
```

```
186             "Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/—
                    ID/bondlengths") {

188             parameters = {

190                 es::ParameterDefinition[results4Id](
191                     "Experiment ID, a component of the operation URL", —
                            IN),
192                 es::ParameterDefinition(
193                     "HTTP response status code", OUT),
194                 es::ParameterDefinition(
195                     "HTTP response message body", OUT)
196             }
197         },
198         es::OperationDefinition[results5](
199             "Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/—
                    ID/cadpac.charges") {

201             parameters = {

203                 es::ParameterDefinition[results5Id](
204                     "Experiment ID, a component of the operation URL", —
                            IN),
205                 es::ParameterDefinition(
206                     "HTTP response status code", OUT),
207                 es::ParameterDefinition(
208                     "HTTP response message body", OUT)
209             }
210         },
211         es::OperationDefinition[results6](
212             "Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/—
                    ID/dmarel.axis") {

214             parameters = {

216                 es::ParameterDefinition[results6Id](
217                     "Experiment ID, a component of the operation URL", —
                            IN),
218                 es::ParameterDefinition(
219                     "HTTP response status code", OUT),
220                 es::ParameterDefinition(
221                     "HTTP response message body", OUT)
222             }
223         },
224         es::OperationDefinition[results7](
225             "Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/—
                    ID/molpak.xyz") {

227             parameters = {

229                 es::ParameterDefinition[results7Id](
230                     "Experiment ID, a component of the operation URL", —
                            IN),
231                 es::ParameterDefinition(
232                     "HTTP response status code", OUT),
233                 es::ParameterDefinition(
234                     "HTTP response message body", OUT)
```

```
235                         }
236                     },
237                     es::OperationDefinition[results8](
238                         ”Results page http://trout1.cs.ucl.ac.uk:18080/axis/polymorph/—
                                ID/pote.dat”) {

240                         parameters = {

242                             es::ParameterDefinition[results8Id](
243                                 ”Experiment ID, a component of the operation URL”, —
                                        IN),
244                             es::ParameterDefinition(
245                                 ”HTTP response status code”, OUT),
246                             es::ParameterDefinition(
247                                 ”HTTP response message body”, OUT)
248                         }
249                     }
250                 }
251             }
252         }

254         clients = {

256             es::ElectronicServiceClientDefinition[csClient](
257                 ”Any computer with an IP address owned by chemistry”) {

259                 owner = PartyDefinition[chemistry]
260             }
261         }

263         behaviours = {

265             es::InformalUsageModeDefinition[anyUsage](
266                 ”Request of any page”) {

268                 operations = {

270                     es::OperationDefinition[static1],
271                     es::OperationDefinition[static2],
272                     es::OperationDefinition[static3],
273                     es::OperationDefinition[submit],
274                     ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke],
275                     es::OperationDefinition[results],
276                     es::OperationDefinition[results1],
277                     es::OperationDefinition[results2],
278                     es::OperationDefinition[results3],
279                     es::OperationDefinition[results4],
280                     es::OperationDefinition[results5],
281                     es::OperationDefinition[results6],
282                     es::OperationDefinition[results7],
283                     es::OperationDefinition[results8]
284                 }
285             }
286             ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                    ModeDefinition[anyHTTPError](
287                 ”Any request results in a code that is not 200”) {
```

```
289                 availabilityClauses = {

291                     ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general]
292                 }

294                 satisfyingConditions = {

296                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            throughput]
297                 }

299                 operations = {

301                     es::OperationDefinition[static1],
302                     es::OperationDefinition[static2],
303                     es::OperationDefinition[static3],
304                     es::OperationDefinition[submit],
305                     ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke],
306                     es::OperationDefinition[results],
307                     es::OperationDefinition[results1],
308                     es::OperationDefinition[results2],
309                     es::OperationDefinition[results3],
310                     es::OperationDefinition[results4],
311                     es::OperationDefinition[results5],
312                     es::OperationDefinition[results6],
313                     es::OperationDefinition[results7],
314                     es::OperationDefinition[results8]
315                 }

317                 usageModes = { es::InformalUsageModeDefinition[anyUsage] }
318             },
319             ::sla1::slang::es::FixedLatencyAvailabilityDependentViolationDependent—
                    FailureModeDefinition
320             [nuisanceDelay](”An annoying delay accessing a page”) {

322                 availabilityClauses = {

324                     ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general]
325                 }

327                 satisfyingConditions = {

329                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            throughput]
330                 }

332                 maxDuration = ::types::Duration(10, S)

334                 operations = {

336                     es::OperationDefinition[static1],
337                     es::OperationDefinition[static2],
338                     es::OperationDefinition[static3],
```

```
339                 es::OperationDefinition[submit],
340                 ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke],
341                 es::OperationDefinition[results],
342                 es::OperationDefinition[results1],
343                 es::OperationDefinition[results2],
344                 es::OperationDefinition[results3],
345                 es::OperationDefinition[results4],
346                 es::OperationDefinition[results5],
347                 es::OperationDefinition[results6],
348                 es::OperationDefinition[results7],
349                 es::OperationDefinition[results8]
350             }

352             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
353         },
354         ::sla1::slang::es::FixedLatencyAvailabilityDependentViolationDependent—
                FailureModeDefinition[seriousDelay](
355             "A serious delay accessing a page that should be treated as a failure.") {

357             availabilityClauses = {

359                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
360             }

362             satisfyingConditions = {

364                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
365             }

367             maxDuration = ::types::Duration(30, S)

369             operations = {

371                 es::OperationDefinition[static1],
372                 es::OperationDefinition[static2],
373                 es::OperationDefinition[static3],
374                 es::OperationDefinition[submit],
375                 ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke],
376                 es::OperationDefinition[results],
377                 es::OperationDefinition[results1],
378                 es::OperationDefinition[results2],
379                 es::OperationDefinition[results3],
380                 es::OperationDefinition[results4],
381                 es::OperationDefinition[results5],
382                 es::OperationDefinition[results6],
383                 es::OperationDefinition[results7],
384                 es::OperationDefinition[results8]
385             }

387             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
388         },
389         ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                ModeDefinition[static1](
```

```
390             "Response is not a valid HTML document defining overall presentation of—
                    polymorph search webclient") {

392             operations = {

394                 es::OperationDefinition[static1]
395             }

397             availabilityClauses = {

399                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
400             }

402             satisfyingConditions = {

404                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
405             }

407             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
408         },
409         ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                ModeDefinition[static2](
410             "Result is not a form allowing the specification of parameters for, and —
                    invocation of the submit page.") {

412             operations = {

414                 es::OperationDefinition[static2]
415             }

417             availabilityClauses = {

419                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
420             }

422             satisfyingConditions = {

424                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
425             }

427             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
428         },
429         ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                ModeDefinition[static3](
430             "Result is not a form allowing the specification of parameters for, and —
                    invocation of the invoke page.") {

432             operations = {

434                 es::OperationDefinition[static3]
435             }
```

```
437             availabilityClauses = {

439                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
440             }

442             satisfyingConditions = {

444                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
445             }

447             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
448         },
449         ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                ModeDefinition[submit](
450             ”Result is not a page acknowledging receipt of submitted parameters”) {

452             operations = {

454                 es::OperationDefinition[submit]
455             }

457             availabilityClauses = {

459                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
460             }

462             satisfyingConditions = {

464                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
465             }

467             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
468         },
469         ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                ModeDefinition[invoke](
470             ”Result is not a page acknowledging the initiation of an experiment”) {

472             operations = {

474                 ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke]
475             }

477             availabilityClauses = {

479                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
480             }

482             satisfyingConditions = {
```

```
484                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
485             }

487             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
488         },
489         ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                ModeDefinition[results](
            ”Result is not a page listing available results”) {

492             operations = {

494                 es::OperationDefinition[results]
495             }

497             availabilityClauses = {

499                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
500             }

502             satisfyingConditions = {

504                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
505             }

507             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
508         },
509         ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                ModeDefinition[results1](
            ”Result is not a valid HTML page summarising all DMAREL executions—
                ”) {

512             operations = {

514                 es::OperationDefinition[results1]
515             }

517             availabilityClauses = {

519                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
520             }

522             satisfyingConditions = {

524                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
525             }

527             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
528         },
```

```
529                 ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                        ModeDefinition[results2](
530                     ”Result is not a valid PNG graphics file representing a scatter graph —
                            summarising all DMAREL executions”) {

532                     operations = {

534                         es::OperationDefinition[results2]
535                     }

537                     availabilityClauses = {

539                         ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                                ConditionClause[general]
540                     }

542                     satisfyingConditions = {

544                         ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                                PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                                throughput]
545                     }

547                     usageModes = { es::InformalUsageModeDefinition[anyUsage] }
548                 },
549                 ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                        ModeDefinition[results3](
550                     ”Result is not a valid XML file containing the results of all DMAREL —
                            executions”) {

552                     operations = {

554                         es::OperationDefinition[results3]
555                     }

557                     availabilityClauses = {

559                         ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                                ConditionClause[general]
560                     }

562                     satisfyingConditions = {

564                         ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                                PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                                throughput]
565                     }

567                     usageModes = { es::InformalUsageModeDefinition[anyUsage] }
568                 },
569                 ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                        ModeDefinition[results4](
570                     ”Result is not the original bondlengths file, uploaded prior to the —
                            commencement of the experiment with the ID specified”) {

572                     operations = {
```

```
574                     es::OperationDefinition[results4]
575                 }

577                 availabilityClauses = {

579                     ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general]
580                 }

582                 satisfyingConditions = {

584                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            throughput]
585                 }

587                 usageModes = { es::InformalUsageModeDefinition[anyUsage] }
588             },
589             ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                    ModeDefinition[results5](
590                 ”Result is not the original cadpak.charges file, uploaded prior to the —
                        commencement of the experiment with the ID specified”) {

592                 operations = {

594                     es::OperationDefinition[results5]
595                 }

597                 availabilityClauses = {

599                     ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general]
600                 }

602                 satisfyingConditions = {

604                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            throughput]
605                 }

607                 usageModes = { es::InformalUsageModeDefinition[anyUsage] }
608             },
609             ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                    ModeDefinition[results6](
610                 ”Result is not the original dmarel.axis file, uploaded prior to the —
                        commencement of the experiment with the ID specified”) {

612                 operations = {

614                     es::OperationDefinition[results6]
615                 }

617                 availabilityClauses = {

619                     ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general]
```

```
620                 }

622                 satisfyingConditions = {

624                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            throughput]
625                 }

627                 usageModes = { es::InformalUsageModeDefinition[anyUsage] }
628             },
629             ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                    ModeDefinition[results7](
630                 ”Result is not the original molpak.xyz file, uploaded prior to the —
                        commencement of the experiment with the ID specified”) {

632                 operations = {

634                     es::OperationDefinition[results7]
635                 }

637                 availabilityClauses = {

639                     ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general]
640                 }

642                 satisfyingConditions = {

644                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            throughput]
645                 }

647                 usageModes = { es::InformalUsageModeDefinition[anyUsage] }
648             },
649             ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailure—
                    ModeDefinition[results8](
650                 ”Result is not the original pote.dat file, uploaded prior to the —
                        commencement of the experiment with the ID specified”) {

652                 operations = {

654                     es::OperationDefinition[results8]
655                 }

657                 availabilityClauses = {

659                     ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general]
660                 }

662                 satisfyingConditions = {

664                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            throughput]
```

```
665             }

667             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
668         },

670         ::combined::slang::es::InformalSuccessModeDefinition[invoke](
671             "Successful initiation of an experiment.") {

673             operations = {

675                 ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke]
676             }

678             incompatibleFailureModes = {

680                 ::sla1::slang::es::InformalAvailabilityDependentViolationDependent—
                        FailureModeDefinition[invoke],
681                 ::sla1::slang::es::FixedLatencyAvailabilityDependentViolation—
                        DependentFailureModeDefinition[seriousDelay]
682             }

684             usageModes = { es::InformalUsageModeDefinition[anyUsage] }
685         },

687         ::combined::slang::es::InformalSuccessModeDefinition[anyResults](
688             "Successful retrieval of any experimental results.") {

690             operations = {

692                 es::OperationDefinition[results1],
693                 es::OperationDefinition[results2],
694                 es::OperationDefinition[results3],
695                 es::OperationDefinition[results4],
696                 es::OperationDefinition[results5],
697                 es::OperationDefinition[results6],
698                 es::OperationDefinition[results7],
699                 es::OperationDefinition[results8]
700             }

702             incompatibleFailureModes = using ::sla1::slang::es {

704                 InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results1],
705                 InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results2],
706                 InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results3],
707                 InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results4],
708                 InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results5],
709                 InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results6],
710                 InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results7],
711                 InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results8]
```

```
712                 }

714                 usageModes = { es::InformalUsageModeDefinition[anyUsage] }
715             },

717             ::sla1::slang::es::FixedLatencyFixedDeadlineDelegatedExecutionDependent—
                    AvailabilityDependentViolationDependentAsynchronousFailureMode—
                    Definition[resultsPageProduction](
718                 "A failure to successfully retrieve all results, without hinderance, within a—
                        week commencing 24 hours after the experiment was started.") {

720                 asynchronousReliabilityClauses = {

722                     ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenalty—
                            MaximalServiceBehaviourRestrictionConditionClause[serious]
723                 }

725                 asynchronousAvailabilityClauses = {

727                     ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general]
728                 }

730                 satisfyingConditions = {

732                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            throughput],
733                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                            PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                            experimentThroughput]
734                 }

736                 operations = {

738                     ::sla1::slang::es::DelegatedExecutionOperationDefinition[invoke]
739                 }

741                 requestOperation =
742                     ::sla1::slang::es::AsynchronousOperationDefinition(
743                         "The request operation") {

745                     operation = ::sla1::slang::es::DelegatedExecutionOperationDefinition—
                            [invoke]

747                     iDParameter = es::ParameterDefinition[invokeId]

749                     successMode = ::combined::slang::es::InformalSuccessMode—
                            Definition[invoke]
750                 }

752                 resultsOperations = {

754                     ::sla1::slang::es::AsynchronousOperationDefinition(
755                         "Results 1") {

757                         operation = es::OperationDefinition[results1]
```

```
759                 iDParameter = es::ParameterDefinition[results1Id]

761                 successMode = ::combined::slang::es::InformalSuccessMode—
                        Definition[anyResults]
762             },
763             ::sla1::slang::es::AsynchronousOperationDefinition(
764                 "Results 2") {

766                 operation = es::OperationDefinition[results2]

768                 iDParameter = es::ParameterDefinition[results2Id]

770                 successMode = ::combined::slang::es::InformalSuccessMode—
                        Definition[anyResults]
771             },
772             ::sla1::slang::es::AsynchronousOperationDefinition(
773                 "Results 3") {

775                 operation = es::OperationDefinition[results3]

777                 iDParameter = es::ParameterDefinition[results3Id]

779                 successMode = ::combined::slang::es::InformalSuccessMode—
                        Definition[anyResults]
780             },
781             ::sla1::slang::es::AsynchronousOperationDefinition(
782                 "Results 4") {

784                 operation = es::OperationDefinition[results4]

786                 iDParameter = es::ParameterDefinition[results4Id]

788                 successMode = ::combined::slang::es::InformalSuccessMode—
                        Definition[anyResults]
789             },
790             ::sla1::slang::es::AsynchronousOperationDefinition(
791                 "Results 5") {

793                 operation = es::OperationDefinition[results5]

795                 iDParameter = es::ParameterDefinition[results5Id]

797                 successMode = ::combined::slang::es::InformalSuccessMode—
                        Definition[anyResults]
798             },
799             ::sla1::slang::es::AsynchronousOperationDefinition(
800                 "Results 6") {

802                 operation = es::OperationDefinition[results6]

804                 iDParameter = es::ParameterDefinition[results6Id]

806                 successMode = ::combined::slang::es::InformalSuccessMode—
                        Definition[anyResults]
807             },
808             ::sla1::slang::es::AsynchronousOperationDefinition(
```

```
809                                 "Results 7") {

811                                 operation = es::OperationDefinition[results7]

813                                 iDParameter = es::ParameterDefinition[results7Id]

815                                 successMode = ::combined::slang::es::InformalSuccessMode—
                                        Definition[anyResults]
816                             },
817                             ::sla1::slang::es::AsynchronousOperationDefinition(
818                                 "Results 8") {

820                                 operation = es::OperationDefinition[results8]

822                                 iDParameter = es::ParameterDefinition[results8Id]

824                                 successMode = ::combined::slang::es::InformalSuccessMode—
                                        Definition[anyResults]
825                             }
826                         }

828                         usageModes = { es::InformalUsageModeDefinition[anyUsage] }

830                         latency = ::types::Duration(24, hr)

832                         deadline = ::types::Duration(7, day)
833                     }
834                 }
835             }
836         }

838         penalties = {

840             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[day1—
                    ](
841                 "Pay 10 pounds sterling within 30 days of administration.") {

843                 amount = 10.0;

845                 deadline = ::types::Duration(30, day)
846             },
847             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[day2—
                    ](
848                 "Pay 20 pounds sterling within 30 days of administration.") {

850                 amount = 20.0;

852                 deadline = ::types::Duration(30, day)
853             },
854             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[day3—
                    ](
855                 "Pay 30 pounds sterling within 30 days of administration.") {

857                 amount = 30.0;

859                 deadline = ::types::Duration(30, day)
860             },
```

```
861             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[day4—
                    ](
862                 "Pay 40 pounds sterling within 30 days of administration.") {

864                 amount = 40.0;

866                 deadline = ::types::Duration(30, day)
867             },
868             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[day5—
                    ](
869                 "Pay 50 pounds sterling within 30 days of administration.") {

871                 amount = 50.0;

873                 deadline = ::types::Duration(30, day)
874             },
875             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[day6—
                    ](
876                 "Pay 60 pounds sterling within 30 days of administration.") {

878                 amount = 60.0;

880                 deadline = ::types::Duration(30, day)
881             },
882             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[—
                    serious](
883                 "Pay 200 pounds sterling within 30 days of administration.") {

885                 amount = 30.0;

887                 deadline = ::types::Duration(30, day)
888             },
889             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[—
                    nuisance1](
890                 "Pay 10 pounds sterling within 30 days of administration.") {

892                 amount = 10.0;

894                 deadline = ::types::Duration(30, day)
895             },
896             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[—
                    nuisance2](
897                 "Pay 10 pounds sterling within 30 days of administration.") {

899                 amount = 30.0;

901                 deadline = ::types::Duration(30, day)
902             },
903             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[—
                    experimentFailure](
904                 "Pay 200 pounds sterling within 30 days of administration.") {

906                 amount = 200.0;

908                 deadline = ::types::Duration(30, day)
909             },
```

```
910             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[per—
                    UseCharge](
911                 "Pay 100 pounds sterling within 30 days of administration.") {

913                 amount = 100.0;

915                 deadline = ::types::Duration(30, day)
916             }
917             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[—
                    termination](
918                 "Pay 2000 pounds sterling within 30 days of administration.") {

920                 amount = 2000.0;

922                 deadline = ::types::Duration(30, day)
923             }

925         }

927         administrationClauses = {

929             ::combined::slang::es::ScheduledConsecutiveAvailabilityAwareReconciliation—
                    AdministrationClause[a1]() {

931                 /* Agreement starts on this date */
932                 administrationStart = ::types::TAIDate(1, 5, 2007)

934                 /* Agreement is administered every friday for three months */
935                 schedule = {

937                     ::combined::slang::PeriodicInterval[fridays]() {

939                         name = "Every friday for the duration of the agreement";

941                         startDate = ::types::TAIDate(1, 5, 2007)

943                         period = ::types::Duration(7, day)

945                         duration = ::types::Duration(1, day)

947                         endDate = ::types::TAIDate(1, 5, 2008)
948                     }
949                 }

951                 accuracyClauses = {

953                     es::PermanentFixedServiceUsageRecordAccuracyClause[a1]() {

955                         dateErrorMargin = ::types::Duration(1, S)

957                         durationErrorMargin = ::types::Duration(1, S)

959                         typeIErrorRate = ::types::Percentage(0.001);

961                         confidence = ::types::Percentage(0.99);
962                     },
```

```
964                 PermanentFixedReportRecordingAccuracyClause[a2]() {

966                     errorMargin = ::types::Duration(1, min)

968                     typeIErrorRate = ::types::Percentage(0.001);

970                     confidence = ::types::Percentage(0.99);
971                 }
972             }

974             conditions = {

976                 /* Availability condition */
977                 ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]() {

979                     reliabilityClauses = {

981                         ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenalty—
                                MaximalServiceBehaviourRestrictionConditionClause[serious]
982                     }

984                     deadline = ::types::Duration(30, min)

986                     usageMode = es::InformalUsageModeDefinition[anyUsage]

988                     penalties = using ::sla1::slang {

990                         SteppedPenalty() {

992                             threshold = ::types::Duration(1, day)

994                             penalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                                    PaymentPenaltyDefinition[day1]
995                         },
996                         SteppedPenalty() {

998                             threshold = ::types::Duration(2, day)

1000                            penalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                                    PaymentPenaltyDefinition[day2]
1001                        },
1002                        SteppedPenalty() {

1004                            threshold = ::types::Duration(3, day)

1006                            penalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                                    PaymentPenaltyDefinition[day3]
1007                        },
1008                        SteppedPenalty() {

1010                            threshold = ::types::Duration(4, day)

1012                            penalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                                    PaymentPenaltyDefinition[day4]
1013                        },
1014                        SteppedPenalty() {
```

```
1016                    threshold = ::types::Duration(5, day)

1018                    penalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                            PaymentPenaltyDefinition[day5]
1019                },
1020                SteppedPenalty() {

1022                    threshold = ::types::Duration(6, day)

1024                    penalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                            PaymentPenaltyDefinition[day6]
1025                }
1026            }
1027        },

1029        /* General input throughput condition */
1030        ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenalty—
                MaximalServiceBehaviourRestrictionConditionClause[throughput]() {

1032            restrictedBehaviours = {

1034                es::InformalUsageModeDefinition[anyUsage]
1035            }

1037            maxOccurrences = 20;

1039            window = ::types::Duration(10, S)
1040        },

1042        /* Serious reliability condition */
1043        ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMaximal—
                ServiceBehaviourRestrictionConditionClause[serious]() {

1045            restrictedBehaviours = using ::sla1::slang::es {

1047                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[anyHTTPError],
1048                FixedLatencyAvailabilityDependentViolationDependentFailureMode—
                        Definition[seriousDelay],
1049                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[static1],
1050                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[static2],
1051                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[static3],
1052                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[submit],
1053                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[invoke],
1054                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results],
1055                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results1],
1056                InformalAvailabilityDependentViolationDependentFailureMode—
                        Definition[results2],
```

```
1057                    InformalAvailabilityDependentViolationDependentFailureMode—
                            Definition[results3],
1058                    InformalAvailabilityDependentViolationDependentFailureMode—
                            Definition[results4],
1059                    InformalAvailabilityDependentViolationDependentFailureMode—
                            Definition[results5],
1060                    InformalAvailabilityDependentViolationDependentFailureMode—
                            Definition[results6],
1061                    InformalAvailabilityDependentViolationDependentFailureMode—
                            Definition[results7],
1062                    InformalAvailabilityDependentViolationDependentFailureMode—
                            Definition[results8]
1063                }

1065                maxOccurrences = 10;

1067                window = ::types::Duration(10, min)

1069                penalty = ::combined::slang::FixedDeadlineFixedPoundsSterlingPayment—
                        PenaltyDefinition[serious]
1070            },

1072            /* Nuisance delay condition */
1073            ::sla1::slang::PermanentFixedWindowFixedOccurrencesSteppedPenalty—
                    MaximalServiceBehaviourRestrictionConditionClause[nuisance]() {

1075                restrictedBehaviours = using ::sla1::slang::es {

1077                    FixedLatencyAvailabilityDependentViolationDependentFailureMode—
                            Definition[nuisanceDelay]
1078                }

1080                maxOccurrences = 10;

1082                window = ::types::Duration(10, min)

1084                penalties = using ::sla1::slang {

1086                    SteppedPenalty() {

1088                        threshold = ::types::Duration(1, day)

1090                        penalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                                PaymentPenaltyDefinition[day1]
1091                    },
1092                    SteppedPenalty() {

1094                        threshold = ::types::Duration(2, day)

1096                        penalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                                PaymentPenaltyDefinition[day2]
1097                    }
1098                }
1099            },

1101            /* Experiment reliability condition */
```

```
1102                ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMaximal—
                        ServiceBehaviourRestrictionConditionClause[experiment]() {

1104                    restrictedBehaviours = using ::sla1::slang::es {

1106                        FixedLatencyFixedDeadlineDelegatedExecutionDependent—
                                AvailabilityDependentViolationDependentAsynchronousFailure—
                                ModeDefinition[resultsPageProduction]
1107                    }

1109                    maxOccurrences = 0;

1111                    window = ::types::Duration(24, hr)

1113                    penalty = ::combined::slang::FixedDeadlineFixedPoundsSterlingPayment—
                            PenaltyDefinition[experimentFailure]
1114                },

1116                /* Experiment throughput condition */
1117                ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenalty—
                        MaximalServiceBehaviourRestrictionConditionClause[experiment—
                        Throughput]() {

1119                    restrictedBehaviours = using ::combined::slang::es {

1121                        InformalSuccessModeDefinition[invoke]
1122                    }

1124                    maxOccurrences = 1;

1126                    window = ::types::Duration(24, hr)
1127                },

1129                /* Experiment charging condition */
1130                ::combined::slang::PermanentFixedWindowFixedOccurrencesFixedPenalty—
                        MinimalServiceBehaviourRestrictionConditionClause[experiment—
                        Charging]() {

1132                    restrictedBehaviours = using ::combined::slang::es {

1134                        InformalSuccessModeDefinition[invoke]
1135                    }

1137                    maxOccurrences = 0;

1139                    window = ::types::Duration(24, hr)

1141                    penalty = ::combined::slang::FixedDeadlineFixedPoundsSterlingPayment—
                            PenaltyDefinition[perUseCharge]
1142                }
1143            }
1144        }

1146        ::combined::slang::es::FixedDeadlineTerminationByReportConsecutiveAvailability—
                AwareReconciliationAdministrationClause[a2]() {

1148            administrationStart = ::types::TAIDate(1, 5, 2007)
```

```
1150                deadline = ::types::Duration(7, day)

1152                accuracyClauses = {

1154                    es::PermanentFixedServiceUsageRecordAccuracyClause[a1],
1155                    PermanentFixedReportRecordingAccuracyClause[a2]
1156                }

1158                conditions = {

1160                    ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailability—
                            ConditionClause[general],
1161                    ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenalty—
                            MaximalServiceBehaviourRestrictionConditionClause[throughput],
1162                    ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMaximal—
                            ServiceBehaviourRestrictionConditionClause[serious],
1163                    ::sla1::slang::PermanentFixedWindowFixedOccurrencesSteppedPenalty—
                            MaximalServiceBehaviourRestrictionConditionClause[nuisance],
1164                    ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMaximal—
                            ServiceBehaviourRestrictionConditionClause[experiment],
1165                    ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenalty—
                            MaximalServiceBehaviourRestrictionConditionClause[experiment—
                            Throughput],
1166                    ::combined::slang::PermanentFixedWindowFixedOccurrencesFixedPenalty—
                            MinimalServiceBehaviourRestrictionConditionClause[experiment—
                            Charging],

1168                    ::combined::slang::FixedPenaltyTerminationByReportConditionClause[—
                            termination]() {

1170                        fixedPenalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                                PaymentPenaltyDefinition[termination]
1171                    }
1172                }
1173            }
1174        }

1176        auxiliaryClauses = {

1178            es::ElectronicServiceInterfaceDefinition[polymorph],
1179            es::ElectronicServiceClientDefinition[csClient],
1180            ::sla1::slang::es::FixedDurationExecutableDefinition[molpak],
1181            ::sla1::slang::es::FixedDurationExecutableDefinition[dmarel]
1182            ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadlineAvailabilityConditionClause[—
                    general],
1183            ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenaltyMaximal—
                    ServiceBehaviourRestrictionConditionClause[throughput],
1184            ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMaximalService—
                    BehaviourRestrictionConditionClause[serious],
1185            ::sla1::slang::PermanentFixedWindowFixedOccurrencesSteppedPenaltyMaximal—
                    ServiceBehaviourRestrictionConditionClause[nuisance],
1186            ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMaximalService—
                    BehaviourRestrictionConditionClause[experiment],
1187            ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenaltyMaximal—
                    ServiceBehaviourRestrictionConditionClause[experimentThroughput],
```

```
1188                ::combined::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMinimal—
                        ServiceBehaviourRestrictionConditionClause[experimentCharging],
1189                ::combined::slang::FixedPenaltyTerminationByReportConditionClause[termination],
1190                es::PermanentFixedServiceUsageRecordAccuracyClause[a1],
1191                PermanentFixedReportRecordingAccuracyClause[a2]
1192                ::combined::slang::PeriodicInterval[fridays],
1193                ::sla1::slang::es::FixedDurationExecutableDefinition[molpak],
1194                ::sla1::slang::es::FixedDurationExecutableDefinition[dmarel]
1195            }

1197        }
1198    }
```

Appendix D

SLA 4: CS and ISP

```
/*
This is an SLA written using the Human−Usable Textual Notation (HUTN) for
the language SLAng.

HUTN is a concrete−syntax standard of the OMG (available http://www.omg.org/)

The SLA is the only object with type ::slang::MutuallyMonitorableSLA.
*/

specification = "http://slang.sourceforge.net/specifications/thesis−combined.emofxmi"

configuration = "http://slang.sourceforge.net/specifications/thesis−combined.hutnxmi"

using ::slang {

    MutuallyMonitorableSLA("SLA between Computer Science and ISP") {

        parties = {

            PartyDefinition[cs](
                "The Department of Computer Science, University College London"),

            PartyDefinition[isp](
                "The ISP")
        }

        services = {

            es::ElectronicServiceDefinition[plotService](
                "The provision of the plotting webservice by the ISP to CS") {

                provider = PartyDefinition[isp]

                client = PartyDefinition[cs]

                interfaces = {

                    es::ElectronicServiceInterfaceDefinition[plotws](
                        "The Plot service located at http://plotws.omii.ac.uk:18080/PlotWS/—
                            services/Graph?wsdl") {

                        owner = PartyDefinition[isp]

                        operations = {
```

```
45                  es::OperationDefinition[makePlot_xyy](
46                      "Operation: makePlot_xyy") {

48                      parameters = {

50                          es::ParameterDefinition("xi", IN),
51                          es::ParameterDefinition("yi", IN),
52                          es::ParameterDefinition("opt", IN),
53                          es::ParameterDefinition("makePlotReturn", OUT)
54                      }
55                  },
56                  es::OperationDefinition[makePlot_xy](
57                      "Operation: makePlot_xy") {

59                      parameters = {

61                          es::ParameterDefinition("xi", IN),
62                          es::ParameterDefinition("yi", IN),
63                          es::ParameterDefinition("opt", IN),
64                          es::ParameterDefinition("makePlotReturn")
65                      }
66                  },
67                  es::OperationDefinition[makePlot_x](
68                      "Operation: makePlot_x") {

70                      parameters = {

72                          es::ParameterDefinition("xi", IN),
73                          es::ParameterDefinition("opt", IN),
74                          es::ParameterDefinition("makePlotReturn")
75                      }
76                  },
77                  es::OperationDefinition[makePlot_xxyy](
78                      "Operation: makePlot_xxyy") {

80                      parameters = {

82                          es::ParameterDefinition("xi", IN),
83                          es::ParameterDefinition("yi", IN),
84                          es::ParameterDefinition("opt", IN),
85                          es::ParameterDefinition("makePlotReturn", OUT)
86                      }
87                  },
88                  es::OperationDefinition[makePlot3D_xyz](
89                      "Operation: makePlot3D_xyz") {

91                      parameters = {

93                          es::ParameterDefinition("xi", IN),
94                          es::ParameterDefinition("yi", IN),
95                          es::ParameterDefinition("yi", IN),
96                          es::ParameterDefinition("opt", IN),
97                          es::ParameterDefinition("makePlotReturn")
98                      }
99                  }
100             }
```

```
101             }
102         }

104         clients = {

106             es::ElectronicServiceClientDefinition[csClient](
107                 ”Any computer with an IP address owned by CS”) {

109                 owner = PartyDefinition[cs]
110             }
111         }

113         behaviours = {

115             es::InformalUsageModeDefinition[anyOperation](
116                 ”Any operation of the service is used”) {

118                 operations = {

120                     es::OperationDefinition[makePlot_xyy],
121                     es::OperationDefinition[makePlot_xy],
122                     es::OperationDefinition[makePlot_x],
123                     es::OperationDefinition[makePlot_xxyy],
124                     es::OperationDefinition[makePlot3D_xyz]
125                 }
126             }

128             ::sla4::slang::es::ScheduledInformalAvailabilityDependentViolation—
                    DependentFailureModeDefinition[failure](
129                 ”The service returns an error code, or the graph returned is inaccurate —
                        with respect to the parameter data or settings”) {

131                 schedule = {

133                     ::combined::slang::PeriodicInterval[operatingHours]() {

135                         name = ”Not midnight until 1 am”;

137                         startDate = ::types::TAIDate(1, 5, 2007) {

139                             hour = 1
140                         }

142                         endDate = ::types::TAIDate(1, 5, 2008)

144                         period = ::types::Duration(24, hr)

146                         duration = ::types::Duration(23, hr)
147                     }
148                 }

150                 operations = {

152                     es::OperationDefinition[makePlot_xyy],
153                     es::OperationDefinition[makePlot_xy],
154                     es::OperationDefinition[makePlot_x],
155                     es::OperationDefinition[makePlot_xxyy],
```

```
156                 es::OperationDefinition[makePlot3D_xyz]
157             }

159             usageModes = { es::InformalUsageModeDefinition[anyOperation] }

161             availabilityClauses = {

163                 ::sla4::slang::es::ScheduledScalingPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
164             }

166             satisfyingConditions = {

168                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
169             }
170         },

172         ::sla4::slang::es::ScheduledFixedLatencyAvailabilityDependentViolation—
                DependentFailureModeDefinition[delay](
173             ”An operation of the service takes longer than 10s to complete”) {

175             schedule = {

177                 ::combined::slang::PeriodicInterval[operatingHours]
178             }

180             maxDuration = ::types::Duration(10, S)

182             operations = {

184                 es::OperationDefinition[makePlot_xyy],
185                 es::OperationDefinition[makePlot_xy],
186                 es::OperationDefinition[makePlot_x],
187                 es::OperationDefinition[makePlot_xxyy],
188                 es::OperationDefinition[makePlot3D_xyz]
189             }

191             usageModes = { es::InformalUsageModeDefinition[anyOperation] }

193             availabilityClauses = {

195                 ::sla4::slang::es::ScheduledScalingPenaltyFixedDeadlineAvailability—
                        ConditionClause[general]
196             }

198             satisfyingConditions = {

200                 ::combined::slang::PermanentFixedWindowFixedOccurrencesNo—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        throughput]
201             }
202         },

204         ::combined::slang::es::InformalSuccessModeDefinition[success](
205             ”Successful production of a graph.”) {
```

```
207                         operations = {

209                             es::OperationDefinition[makePlot_xyy],
210                             es::OperationDefinition[makePlot_xy],
211                             es::OperationDefinition[makePlot_x],
212                             es::OperationDefinition[makePlot_xxyy],
213                             es::OperationDefinition[makePlot3D_xyz]
214                         }

216                         incompatibleFailureModes = {

218                             ::sla4::slang::es::ScheduledInformalAvailabilityDependentViolation—
                                    DependentFailureModeDefinition[failure],
219                             ::sla4::slang::es::ScheduledFixedLatencyAvailabilityDependent—
                                    ViolationDependentFailureModeDefinition[delay]
220                         }

222                         usageModes = { es::InformalUsageModeDefinition[anyOperation] }
223                     }
224                 }
225             }
226         }

228         penalties = {

230             ::sla4::slang::FixedDeadlineScalingPoundsSterlingPaymentPenaltyDefinition[failure](
231                 "Pay 1 pound per hour of failures or unavailability.") {

233                 amountPerHour = 1.0;

235                 deadline = ::types::Duration(30, day)
236             },
237             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[—
                    termination](
238                 "Pay 100 pounds on termination.") {

240                 amount = 100.0;

242                 deadline = ::types::Duration(30, day)
243             }
244             ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition[per—
                    UseCharge](
245                 "Pay 5 pence per plot.") {

247                 amount = 0.05;

249                 deadline = ::types::Duration(30, day)
250             }

252         }

254         administrationClauses = {

256             ::combined::slang::es::ScheduledConsecutiveAvailabilityAwareReconciliation—
                    AdministrationClause[a1]() {
```

```
258                 /* Agreement starts on this date */
259                 administrationStart = ::types::TAIDate(1, 5, 2007)

261                 /* Agreement is administered every friday for three months */
262                 schedule = {

264                     ::combined::slang::PeriodicInterval[fridays]() {

266                         name = "Every friday for the duration of the agreement";

268                         startDate = ::types::TAIDate(1, 5, 2007)

270                         period = ::types::Duration(7, day)

272                         duration = ::types::Duration(1, day)

274                         endDate = ::types::TAIDate(1, 5, 2008)
275                     }
276                 }

278                 accuracyClauses = {

280                     es::PermanentFixedServiceUsageRecordAccuracyClause[a1]() {

282                         dateErrorMargin = ::types::Duration(1, S)

284                         durationErrorMargin = ::types::Duration(1, S)

286                         typeIErrorRate = ::types::Percentage(0.001);

288                         confidence = ::types::Percentage(0.99);
289                     },

291                     PermanentFixedReportRecordingAccuracyClause[a2]() {

293                         errorMargin = ::types::Duration(1, min)

295                         typeIErrorRate = ::types::Percentage(0.001);

297                         confidence = ::types::Percentage(0.99);
298                     }
299                 }

301                 conditions = {

303                     /* Availability condition */
304                     ::sla4::slang::es::ScheduledScalingPenaltyFixedDeadlineAvailabilityCondition—
                            Clause[general]() {

306                         schedule = {

308                             ::combined::slang::PeriodicInterval[operatingHours]
309                         }

311                         reliabilityClauses = {
```

```
313                 ::sla4::slang::PermanentFixedWindowFixedOccurrencesScaling—
                        PenaltyMaximalServiceBehaviourRestrictionConditionClause[—
                        failures]
314             }

316             deadline = ::types::Duration(30, min)

318             usageMode = es::InformalUsageModeDefinition[anyOperation]

320             penalty = ::sla4::slang::FixedDeadlineScalingPoundsSterlingPayment—
                    PenaltyDefinition[failure]
321         },

323         /* General input throughput condition */
324         ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenalty—
                MaximalServiceBehaviourRestrictionConditionClause[throughput]() {

326             restrictedBehaviours = {

328                 es::InformalUsageModeDefinition[anyOperation]
329             }

331             maxOccurrences = 20;

333             window = ::types::Duration(10, S)
334         },

336         /* Delays and failures condition */
337         ::sla4::slang::PermanentFixedWindowFixedOccurrencesScalingPenalty—
                MaximalServiceBehaviourRestrictionConditionClause[failures]() {

339             restrictedBehaviours = using ::sla4::slang::es {

341                 ScheduledInformalAvailabilityDependentViolationDependentFailure—
                        ModeDefinition[failure],
342                 ScheduledFixedLatencyAvailabilityDependentViolationDependent—
                        FailureModeDefinition[delay]
343             }

345             maxOccurrences = 10;

347             window = ::types::Duration(10, min)

349             penalty = ::sla4::slang::FixedDeadlineScalingPoundsSterlingPayment—
                    PenaltyDefinition[failure]
350         }

352         /* Charging condition */
353         ::combined::slang::PermanentFixedWindowFixedOccurrencesFixedPenalty—
                MinimalServiceBehaviourRestrictionConditionClause[experiment—
                Charging]() {

355             restrictedBehaviours = using ::sla4::slang::es {

357                 ::combined::slang::es::InformalSuccessModeDefinition[success]
358             }
```

```
360                         maxOccurrences = 0;

362                         window = ::types::Duration(24, hr)

364                         penalty = ::combined::slang::FixedDeadlineFixedPoundsSterlingPayment—
                                PenaltyDefinition[perUseCharge]
365                     }

367                 }
368             }

370             ::combined::slang::es::FixedDeadlineTerminationByReportConsecutiveAvailability—
                    AwareReconciliationAdministrationClause[a2]() {

372                 administrationStart = ::types::TAIDate(1, 5, 2007)

374                 deadline = ::types::Duration(7, day)

376                 accuracyClauses = {

378                     es::PermanentFixedServiceUsageRecordAccuracyClause[a1],
379                     PermanentFixedReportRecordingAccuracyClause[a2]
380                 }

382                 conditions = {

384                     ::sla4::slang::es::ScheduledScalingPenaltyFixedDeadlineAvailabilityCondition—
                            Clause[general],
385                     ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenalty—
                            MaximalServiceBehaviourRestrictionConditionClause[throughput],
386                     ::sla4::slang::PermanentFixedWindowFixedOccurrencesScalingPenalty—
                            MaximalServiceBehaviourRestrictionConditionClause[failures],
387                     ::combined::slang::FixedPenaltyTerminationByReportConditionClause[—
                            termination]() {

389                         fixedPenalty = ::combined::slang::FixedDeadlineFixedPoundsSterling—
                                PaymentPenaltyDefinition[termination]
390                     }
391                 }
392             }
393         }

395         auxiliaryClauses = {

397             es::ElectronicServiceInterfaceDefinition[plotws],
398             es::ElectronicServiceClientDefinition[csClient],
399             ::sla4::slang::es::ScheduledScalingPenaltyFixedDeadlineAvailabilityConditionClause[—
                    general],
400             ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenaltyMaximal—
                    ServiceBehaviourRestrictionConditionClause[throughput],
401             ::sla4::slang::PermanentFixedWindowFixedOccurrencesScalingPenaltyMaximal—
                    ServiceBehaviourRestrictionConditionClause[failures],
402             ::combined::slang::FixedPenaltyTerminationByReportConditionClause[termination],
403             ::combined::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMinimal—
                    ServiceBehaviourRestrictionConditionClause[experimentCharging],
404             es::PermanentFixedServiceUsageRecordAccuracyClause[a1],
405             PermanentFixedReportRecordingAccuracyClause[a2] ,
```

```
406                 ::combined::slang::PeriodicInterval[fridays],
407                 ::combined::slang::PeriodicInterval[operatingHours]
408             }
409         }
410 }
```

Appendix E

Specification - Combined

E.1 Package - ::types

Informal: Contains types used in both the syntactic and semantic model.

E.1.1 Enumeration - ::types::TimeUnit

Definitive: An enumeration type used to indicate a unit of time associated with some quantity in the model.

- S

  Definitive: Seconds.

- mS

  Definitive: Milli-seconds.

- nS

  Definitive: Nano-seconds.

- min

  Definitive: Minutes.

- hr

  Definitive: Hours.

- day

  Definitive: Days (24 hours).

E.1.2 Class - ::types::Percentage

Definitive: In the syntactic model, indicates a percentage written in an SLA. In the services model, this is the type of features of an object that can be interpreted as a degree of completeness of some totality.

Properties:

- value : ::types::Real

  Definitive: The percentage is this value multiplied by 100.

Operations:

- No operations.

Invariants:

- Wellformedness: Percentages are expressed as a value greater than 0.

```
value >= 0
```

E.1.3 Class - ::types::Duration

Definitive: In the syntactic model a duration is the specification of a length of time. In the services model, a duration is either an actual length of time, or a record of a length of time.

Properties:

- value : ::types::Real

  Definitive: Interpreted as a number of units of the type specified in the unit property of the duration object, the value is the length of the duration.

- unit : ::types::TimeUnit

  Definitive: The time unit, by which the value of this duration may be interpreted as an actual duration.

Operations:

- inMs() : ::types::Real

  Informal: Converts this duration to a number of milliseconds.

  Evaluates to:

```
if unit = TimeUnit.mS then value
  else
    if unit = TimeUnit.nS then value / 1000
    else
      if unit = TimeUnit.S then value * 1000
      else
        if unit = TimeUnit.min then value * 1000 * 60
        else
          if unit = TimeUnit.hr then value * 1000 * 60 * 60
          else
            value * 1000 * 60 * 60 * 24
          endif
        endif
      endif
    endif
  endif
```

- eq(s : ::types::Duration) : ::types::Boolean

  Informal: Defines non-object equality for duration objects.

  Evaluates to:

```
inMs() = s.inMs()
```

Invariants:

- Wellformedness: Durations should never be negative.

```
not (value < 0)
```

E.1.4 Abstract class - ::types::Date

Definitive: In the syntactic model a date is the specification of an instant in time. In the services model, a duration is either an actual instant time, or a record of an instant of time.

Properties:

- No properties.

Operations:

- inMs() : ::types::Real

  Informal: (abstract) Converts this date to the number of milliseconds that the date is after 00:00 Jan 1, 2000, UTC+0.

- eq(d : ::types::Date) : ::types::Boolean

  Informal: (abstract) Defines non-object equality for date objects.

Invariants:

- No invariants.

E.1.5 Class - ::types::TAIDate

Extends: ::types::Date, pg. 328

Definitive: A date according to International Atomic Time. This does not accomodate leap seconds (because we cannot predict what leap seconds will be needed in the future).

Properties:

- year : ::types::Integer

  Definitive: The contemporary era year, 2000 or later, in which this date occurs.

- month : ::types::Integer

  Definitive: The month, from 1 to 12, in which this date occurs.

- day : ::types::Integer

  Definitive: The day of the month in which this date occurs.

- hour : ::types::Integer

  Definitive: The hour within which this date occurs.

- minute : ::types::Integer

  Definitive: The minute upon which this date occurs.

- second : ::types::Real

  Definitive: The second and fractional seconds within the minute upon which this date occurs.

Operations:

- inMs() : ::types::Real

  Informal: Returns the number of milliseconds to this date counting from 00:00:00.000, 1/1/2000.

  Evaluates to:

```
let yearsSince = year - 2000
in
let leapYearsSince = priorLeapYears()
in
let nonLeapYearsSince = yearsSince - leapYearsSince
in
(leapYearsSince * 366 * 24 * 60 * 6e+4) +
(nonLeapYearsSince * 365 * 24 * 60 * 6e+4) +
(dayInYear() * 24 * 60 * 6e+4) +
(hour * 60 * 6e+4) +
(minute * 6e+4) +
(second * 1e+3)
```

- isLeapYear() : ::types::Boolean

  Informal: Returns true if this date occurs within a leap year. Returns false otherwise.

  Evaluates to:

```
year.mod(4) = 0
and
year.mod(400) <> 0
```

- priorLeapYears() : ::types::Real

  Informal: Returns the number of leap-years occuring between the year in which this date occurs and 2000.

  Evaluates to:

```
let yearsSince = (year - 1) - 2000
in
(yearsSince / 4).floor() -
(yearsSince / 400).floor()
```

- daysInMonth() : ::types::Integer[0, *] ordered

  Informal: Returns a sequence listing the number of days in each month in the year within which this date occurs.

  Evaluates to:

```
Sequence(Integer) {
  31,
  if isLeapYear() then 29 else 28 endif,
  31,
  30,
  31,
  30,
  31,
  31,
  30,
  31,
  30,
  31
}
```

- dayInYear() : ::types::Integer

  Informal: Returns the day (counted from the 1st of January) in the year upon which this date occurs.

  Evaluates to:

```
if(month = 1) then day
else
  daysInMonth()->subSequence(1, month - 1)->sum() + day
endif
```

Invariants:

- Informal: We only deal with dates after 2000. All parameters of the date must be within normal ranges.

```
year >= 2000
and
month >= 1
and
month <= 12
and
day >= 1
and
day <= daysInMonth()->at(month)
and
hour >= 0
and
hour <= 23
and
minute >= 0
and
minute <= 59
```

```
and
second >= 0
and
second < 60
```

E.1.6 Primitive type - ::types::Real

Definitive: In the syntactic model, indicates real numbers written into SLAs. In the service model, this is the type for attributes of an object that can be interpreted as having a value within a continous range.

Equivalent to the OCL real type.

E.1.7 Primitive type - ::types::Boolean

Definitive: In the syntactic model, indicates a value of true or false written into SLAs. In the service model, this is the type for attributes of an object that can be interpreted as being either true or false.

Equivalent to the OCL boolean type.

E.1.8 Primitive type - ::types::Integer

Definitive: In the syntactic model, indicates an whole number written into an SLA. In the service model, this is the type for attributes of an object that can be interpreted as being a natural quantity.

Equivalent to the OCL integer type.

E.1.9 Primitive type - ::types::String

Definitive: In the syntactic model, indicates some text included in an SLA. In the service model, indicates some information present in the domain.

Equivalent to the OCL string type.

E.2 Package - ::slang

Informal: The slang package contains type specifications for SLAs expressible in the SLAng language and their component expressions. Subpackages contain types specific to particular kinds of SLA, for example electronic service SLAs.

E.2.1 Abstract class - ::slang::AccuracyClause

Extends: ::slang::AuxiliaryClause, pg. 336

Definitive: Defines accuracy conditions over the evidence submitted to administration clauses.

This clause is abstract because rules for determining what evidence should be considered, and how it should be determined to be acceptably accurate need to be specified.

Properties:

- administrationClauses : ::slang::AdministrationClause[1, *] unique

  Opposite: ::slang::AdministrationClause.accuracyClauses : ::slang::AccuracyClause[0, *] unique

  Definitive: Accuracy clauses are associated with administration clauses in order to require accuracy of the evidence submitted during administration.

- typeIErrorRate : ::types::Percentage

  Definitive: The likelihood that a report gathered honestly according to accuracy constraints defined for all uncertain parameters will contain an unacceptable number of errors.

  Informal: This parameter effectively sets a limit on the number of errors that can included in an report, either through dishonesty or by accident. See the invariant below for a fuller discussion of the effect of this parameter.

  A very small value should be chosen for this property of the SLA. A significant degree of cheating in an account will rapidly make the account highly unlikely. However, the probability of seeing at least one reasonably unlikely account in a set of accounts associated with an SLA rises with the size of the set. This likelihood should therefore be kept low to avoid unnecessary disagreements in the event of occasional unlikely accounts being submitted in good faith.

  In the event that the invariant associated with this parameter is violated, the SLA is invalidated and the parties will have to take whatever action they deem necessary.

- confidence : ::types::Percentage

  Definitive: Confidence that any measured value falls within its expected error margins.

Operations:

- getMeasurementCount(evidence : ::services::Evidence[0, *] unique) : ::types::Integer

  Informal: (abstract) Count the number of measurements covered by this clause for a given administration.

- getAccurateMeasurementCount(evidence : ::services::Evidence[0, *] unique) : ::types::Integer

  Informal: (abstract) Count the number of accurate measurements covered by this clause for a given administration.

  Note that in real life this can never be evaluated with certainty. However the accuracy constraint overall can be approximately monitored using a statistical hypothesis test.

- fact(n : ::types::Integer) : ::types::Integer

  Informal: Calculates the factorial of a positive integer, or -1 otherwise.

  Evaluates to:

  ```
  if n = 0
  then 1
  else
    if n < 0
    then -1
    else n * fact(n - 1)
    endif
  endif
  ```

- pick(n : ::types::Integer,
  r : ::types::Integer) : ::types::Integer

  Informal: Calculates the number of ways to chose r objects from n possibilities in a particular order

  Evaluates to:

  ```
  if r > 1 then
    (n - (r - 1)) * pick(n, r - 1)
  else n
  endif
  ```

- choose(n : ::types::Integer,
  r : ::types::Integer) : ::types::Integer

  Informal: Calculates the number of ways to choose r objects from n possibilities in no particular order

  Evaluates to:

  ```
  pick(n, r) div fact(r)
  ```

- raise(value : ::types::Real,
  power : ::types::Integer) : ::types::Real

  Informal: Raise a value to an integer power

  Evaluates to:

  ```
  if power = 0 then 1.0
  else
    value * raise(value, power - 1)
  endif
  ```

- errorCountProbability(n : ::types::Integer,
  r : ::types::Integer) : ::types::Real

  Informal: Calculate the probability of seeing r errors in n measurements.

  Evaluates to:

```
choose(n, r) * raise(1 - confidence.value, r) *
  raise(confidence.value, n - r)
```

- findD(sum : ::types::Real,
  n : ::types::Integer,
  d : ::types::Integer) : ::types::Integer

  Informal: Determines the threshold number of violations that can be tolerated for a log of size n. Recursive.

  Evaluates to:

```
if d = 0 then 0
else
  let p = errorCountProbability(n, d) in
  if sum > typeIErrorRate.value
  then -1
  else
    if sum + p > typeIErrorRate.value
    then d
    else findD(sum + p, n, d - 1)
    endif
  endif
endif
```

- getMaximumAcceptableErrors(measurementCount : ::types::Integer) : ::types::Integer

  Informal: Calculates the maximum number of acceptable errors in an account of the specified size, assuming the confidence in the measurements is as specified in the SLA, and given the target typeIErrorRate.

  Evaluates to:

```
findD(0.0, measurementCount, measurementCount)
```

- evidenceIsAccurate(evidence : ::services::Evidence[0, *] unique) : ::types::Boolean

  Informal: Assesses whether some set of evidence is accurate according to this clause.

  Evaluates to:

```
let n = getMeasurementCount(evidence)
in
n - getAccurateMeasurementCount(evidence) <=
  getMaximumAcceptableErrors(n)
```

Invariants:

- No invariants.

E.2.2 Abstract class - ::slang::AdministrationClause

Definitive: An administrative clause defines the circumstances under which administration of the SLA should occur.

This class is abstract because rules concerning when the SLA should be administered must be specified (as invariants of subclasses, related to the events of the services underlying the.

Properties:

- accuracyClauses : ::slang::AccuracyClause[0, *] unique

  Opposite: ::slang::AccuracyClause.administrationClauses : ::slang::AdministrationClause[1, *] unique

  Definitive: Administration clauses reference a set of accuracy clauses that determine the required accuracy of evidence submitted during administration.

- conditions : ::slang::ConditionClause[1, *] unique

  Opposite: ::slang::ConditionClause.administrationClauses : ::slang::AdministrationClause[1, *] unique

  Definitive: Administration clauses reference a set of condition clauses that calculate violations based on the evidence agreed during administration.

- sLA : ::slang::SLA

  Opposite: ::slang::SLA.administrationClauses : ::slang::AdministrationClause[0, *] unique ordered

  Definitive: Administration clauses form part of an SLA.

- administrations : ::services::Administration[0, *] unique

  Opposite: ::services::Administration.administrationClause : ::slang::AdministrationClause

  Definitive: Administration clauses may trigger administrations.

Operations:

- eventRelevant(administration : ::services::Administration,
  event : ::services::Event) : ::types::Boolean

  Informal: (abstract) Determines whether some event is relevant to a particular administration.

- administered() : ::types::Boolean

  Informal: (abstract) Checks whether a set of events includes the correct administration of this clause.

- sLAEvents() : ::services::Event[0, *] unique

  Informal: Administration clauses identify administrations and events pertinent to the conditions that they define as being events pertinent to the SLA.

  Evaluates to:

  ```
  conditions.sLAEvents()->union(administrations)->asSet()
  ```

- services() : ::slang::ServiceDefinition[0, *] unique

  Informal: Identifies the services relevant to this administration clause.

  Evaluates to:

  ```
  conditions->collect(
    if service().oclIsUndefined()
    then Set(::slang::ServiceDefinition) {}
    else Set(::slang::ServiceDefinition) { service() }
    endif
  )->asSet()
  ```

Invariants:

- Wellformedness: The monitoring obligation: If an event is relevant to an administration associated with this clause then adequate evidence to administer all conditions associated with the clause must be provided by the participants collectively in the agreed account associated with the administration.

```
let
events = sLA.events
in
administrations->forall(a : ::services::Administration |

  events->forall(e : ::services::Event |

    sLAEvents()->includes(e)
    and
    eventRelevant(a, e)
    implies
    conditions->forall(c : ConditionClause |

      c.sLAEvents()->includes(e)
      implies
      c.evidenced(e, a)
    )
  )
)
```

- Wellformedness: All condition and accuracy clauses associated with this clause are in the same SLA as this clause.

```
conditions->forall(c : ConditionClause |

  c.sLA = sLA
)
and
accuracyClauses->forall(a : AccuracyClause |

  a.sLA = sLA
)
```

- Wellformedness: The accuracy constraint: All accounts submitted in administrations of this clause must be accurate according to all accuracy clauses associated with this clause.

```
accuracyClauses->forall(
  aC : AccuracyClause |

  administrations.submittedEvidence->forall(
    a : ::services::Account |


    aC.evidenceIsAccurate(a.evidence)
  )
)
```

- Wellformedness: The penalty calculation obligation: Violations relating to evidence agreed in administrations associated with this clause must be calculated according to the condition clauses associated with this clause.

```
administrations->forall(a : ::services::Administration |

  conditions->forall(violationsCalculated(a))
)
```

- Welformedness: Administrations should always be visible to both the client and provider of any services administered.

```
administrations->forall(a : ::services::Administration |

  a.witnesses->includesAll(services().provider.party)
  and
  a.witnesses->includesAll(services().client.party)
)
```

E.2.3 Abstract class - ::slang::AuxiliaryClause

Definitive: Auxiliary definitions are used to describe things significant to an SLA in addition to the definitions of services, parties and penalties.

Properties:

- sLA : ::slang::SLA

  Opposite: ::slang::SLA.auxiliaryClauses : ::slang::AuxiliaryClause[0, *] unique ordered

  Definitive: Auxiliary definitions form part of an SLA.

Operations:

- No operations.

Invariants:

- No invariants.

E.2.4 Abstract class - ::slang::ConditionClause

Extends: ::slang::AuxiliaryClause, pg. 336

Definitive: A condition clause relates bad behaviour of the service to the payment of penalties, and forms part of the conditions of an SLA. It may make reference to terms defined in the SLA.

This clause is abstract because the rules and parameters governing the calculation of violations must be specified.

Properties:

- administrationClauses : ::slang::AdministrationClause[1, *] unique

  Opposite: ::slang::AdministrationClause.conditions : ::slang::ConditionClause[1, *] unique

  Definitive: Condition clauses apply to administration clauses and define the violations that should be calculated in administrations associated with those administration clauses.

Operations:

- priorClauses() : ::slang::ConditionClause[0, *] unique ordered

  Informal: Gets the ordered set of condition clauses preceding this condition clause in the SLA. Can be used to enforce an evaluation order for clauses.

  Evaluates to:

```
let conditions = sLA.auxiliaryClauses->select(
  oclIsKindOf(ConditionClause))
in
conditions->subOrderedSet(1, conditions->indexOf(self))->collect(
  oclAsType(ConditionClause))->asOrderedSet()
```

- sLAEvents() : ::services::Event[0, *] unique

  Informal: (abstract) Condition clauses establish the relevance of certain events to the sLA that contains them.

- service() : ::slang::ServiceDefinition[0, 1]

  Informal: (abstract) The service over which this clause places conditions.

- evidenced(event : ::services::Event,
  administration : ::services::Administration) : ::types::Boolean

  Informal: (abstract) Determines whether adequate evidence exists for an event within the agreed evidence presented in an administration, to determine violations of these conditions.

- violationsCalculated(administration : ::services::Administration) : ::types::Boolean

  Informal: (abstract) Check that administrations have correctly calculated violations associated with this clause.

Invariants:

- No invariants.

E.2.5 Abstract class - ::slang::Definition

Definitive: The terms of an SLA contain a number of definitions of various types of things in the real world.

Properties:

- identifier : ::types::String[0, 1]

  Definitive: Definitions may be given an identifying string to allow convenient reference to made to them outside the context of the SLA. The form and content of the identifying string are unconstrained. The identifying string primarily identifies the definition, not the object described by the definition. Therefore the definition should be construed based on the contents of the description field only.

- description : ::types::String

  Definitive: A description of the thing in the real world being defined. Things associated with this definition must be compatible with the description given for them. The parties to any SLA should ensure before entering the SLA that all terms are defined unambiguously and to their satisfaction. At the agreement of the parties, descriptions included in the SLA may be unambiguous references to descriptions of things maintained externally to the SLA. For example, if the SLA is embedded in a document describing a larger service provision agreement, the SLA may refer to a definition of the service in question contained in the larger document, external to the SLA.

Operations:

- No operations.

Invariants:

- No invariants.

E.2.6 Class - ::slang::MutuallyMonitorableSLA

Extends: ::slang::SLA, pg. 348

Definitive: Some types of SLA are administered, meaning that the client and provider consult on the evidence upon which the determination of violations will be based. SLAs that are mutually monitorable, but not arbitratable by a third party may need to be administered in order to maintain trust relationships between the parties.

Properties:

- No properties.

Operations:

- No operations.

Invariants:

- Wellformedness: All service events of relevance to the SLA must be monitorable by both the client and provider of the service.

```
administrationClauses->forall(a : AdministrationClause |

  a.services()->forall(s : ServiceDefinition |

    a.sLAEvents()->forall(

      witnesses->includes(s.provider.party)
      and
      witnesses->includes(s.client.party)
    )
  )
)
```

E.2.7 Class - ::slang::PartyDefinition

Extends: ::slang::Definition, pg. 337

Definitive: A definition of some person or organisation with a role to play in the service provision scenario.

Properties:

- sLA : ::slang::SLA

  Opposite: ::slang::SLA.parties : ::slang::PartyDefinition[2, *] unique

  Definitive: A party definition forms part an SLA.

- party : ::services::Party

  Definitive: A party definition describes some real world party.

Operations:

- No operations.

Invariants:

- No invariants.

E.2.8 Class - ::slang::PenaltyDefinition

Extends: ::slang::Definition, pg. 337

Definitive: A penalty definition is a pre-agreed penalty that some party will have to pay if a violation of a particular type occurs

Properties:

- violations : ::services::Violation[0, *]

  Opposite: ::services::Violation.penalty : ::slang::PenaltyDefinition[0, 1]

  Definitive: A penalty definition may be cited as being payable in the event of a violation being discovered.

- sLA : ::slang::SLA

  Opposite: ::slang::SLA.penalties : ::slang::PenaltyDefinition[0, *] unique

  Definitive: A penalty definition is part of some SLA.

Operations:

- No operations.

Invariants:

- No invariants.

E.2.9 Class - ::slang::PermanentFixedReportRecordingAccuracyClause

Extends: ::slang::ReportRecordingAccuracyClause, pg. 340

Definitive: A termination report accuracy clause that always applies and specifies a minimum accuracy for evidence relating to the exchange of termination reports.

Informal: If termination-by-report conditions are used, a clause of this kind must be included in an SLA to ensure that evidence related to termination reports is accurate.

Properties:

- errorMargin : ::types::Duration

  Definitive: This is the error margin that must be met when reporting the delivery time of a termination report with the specified confidence and type I error rate, according to this clause.

Operations:

- calculateErrorMargin(report : ::services::Report,
  evidence : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: Get the error margin permitted for a termination report.

  Evaluates to:

  ```
  errorMargin.inMs()
  ```

Invariants:

- No invariants.

E.2.10 Abstract class - ::slang::ReconciliationAdministrationClause

Extends: ::slang::AdministrationClause, pg. 333

Definitive: An administration clause for a mutually monitorable SLA, obliging clients and providers of relevant services to report on all events.

Properties:

- No properties.

Operations:

- No operations.

Invariants:

- Wellformedness: The client and provider participation obligation. Clients and providers of all services over which the clause places conditions must participate in all administrations of this clause.

  ```
  administrations->forall(a : ::services::Administration |

    a.participants->includesAll(conditions.service().client.party)
    and
    a.participants->includesAll(conditions.service().provider.party)
  )
  ```

- Wellformedness: The reconciliation obligation: clients and providers of all services being administered must support all agreed evidence.

  ```
  administrations->forall(a : ::services::Administration |

    services()->forall(s : ServiceDefinition |

      a.agreed->forall(e : ::services::Evidence |
  ```

```
        e.supporters->includes(s.provider.party)
        and
        e.supporters->includes(s.client.party)
      )
    )
  )
```

E.2.11 Abstract class - ::slang::ReportRecordingAccuracyClause

Extends: ::slang::AccuracyClause, pg. 331

Definitive: These clauses enforce accuracy constraints on the reporting of the date of termination reports.

This clause is abstract because the error margin for delivery time must be specified.

Properties:

- No properties.

Operations:

- calculateErrorMargin(report : ::services::Report,
  evidence : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: (abstract) Get the error margin permitted for a termination report.

- getMeasurementCount(evidence : ::services::Evidence[0, *] unique) : ::types::Integer

  Informal: Count the number of measurements covered by this clause for a given administration.

  Evaluates to:

```
if evidence->exists(e : ::services::Evidence |

  e.oclIsKindOf(::services::ReportRecord)
  and
  e.oclAsType(::services::ReportRecord).report.oclIsKindOf(
    ::services::TerminationReport)
)
then 1
else 0
endif
```

- getAccurateMeasurementCount(evidence : ::services::Evidence[0, *] unique) : ::types::Integer

  Informal: Count the number of accurate measurements covered by this clause for a given administration.

  Note that in real life this can never be evaluated with certainty. However the accuracy constraint overall can be approximately monitored using a statistical hypothesis test.

  Evaluates to:

```
if evidence->exists(e : ::services::Evidence |

  e.oclIsKindOf(::services::ReportRecord)
  and
  (
    let record = e.oclAsType(::services::ReportRecord)
    in
    let report = record.report.oclAsType(
      ::services::Report)
    in
    report.date.inMs() <= record.date.inMs() +
      calculateErrorMargin(report, evidence)
```

```
      and
      report.date.inMs() >= record.date.inMs() -
        calculateErrorMargin(report, evidence)
    )
  )
  then 1
  else 0
  endif
```

Invariants:

- No invariants.

E.2.12 Abstract class - ::slang::ServiceBehaviourDefinition

Extends: ::slang::Definition, pg. 337

Definitive: A description of a service behaviour, identifying the events that constitute an instance of the behaviour and the party considered responsible for causing the behaviour to occur.

Properties:

- service : ::slang::ServiceDefinition

  Opposite: ::slang::ServiceDefinition.behaviours : ::slang::ServiceBehaviourDefinition[0, *] unique

  Definitive: A service behaviour definition forms part of a service definition.

Operations:

- calculateResponsibleParty() : ::slang::PartyDefinition

  Informal: (abstract) Get the party held responsible for all instances of this type of behaviour.

- sLAEvents() : ::services::Event[0, *] unique

  Informal: (abstract) Behaviour definitions establish the relevance of certain events to the sLA that contains them.

- evidenced(event : ::services::Event,
  administration : ::services::Administration) : ::types::Boolean

  Informal: (abstract) Determines whether adequate evidence exists for an event to determine whether this behaviour has occurred.

- getFirstInstanceOf(evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: (abstract) Given a set of events, find the first instance of the described behaviour (or the empty set if the behaviour does not occur), in the specified set of evidence, a subset of the agreed evidence of the specified administration.

- getNextInstanceAfter(prior : ::services::Evidence[0, *] unique,
  evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: (abstract) Given a set of events and a subset of these events representing a an instance of the behaviour described, find another instance of the behaviour. Iteration across these behaviours instances should cover all instances of the behaviour in the specified set of events.

- getBehaviourTime(behaviour : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: (abstract) Get the notional time that a behaviour is deemed to have occurred.

Invariants:

- No invariants.

E.2.13 Abstract class - ::slang::ServiceBehaviourRestrictionConditionClause

Extends: ::slang::ConditionClause, pg. 336

Definitive:

Properties:

- restrictedBehaviours : ::slang::ServiceBehaviourDefinition[1, *] unique

  Definitive: The service behaviours associated with this clause.

Operations:

- calculateMaxOccurrences(date : ::types::Real,
  administration : ::services::Administration) : ::types::Integer

  Informal: (abstract) The maximum number of occurrences of the behaviour that may be observed within the sliding window starting at the time specified (in mS).

- calculateWindow(date : ::types::Real,
  administration : ::services::Administration) : ::types::Real

  Informal: (abstract) The width of a notional sliding time window, starting at the time specified, within which no more than maxOccurances of the restricted behaviours may occur.

- sLAEvents() : ::services::Event[0, *] unique

  Informal: The events relevant to a service behaviour restriction clause are the events relevant to the behaviours it restricts.

  Evaluates to:

  ```
  restrictedBehaviours.sLAEvents()->asSet()
  ```

- service() : ::slang::ServiceDefinition

  Informal: The services over which this clause places conditions are the services over which the behaviours are described.

  Evaluates to:

  ```
  restrictedBehaviours.service->any(true)
  ```

- evidenced(event : ::services::Event,
  administration : ::services::Administration) : ::types::Boolean

  Informal: An event is evidenced for a behaviour restriction if it is evidenced for all of the restricted behaviours to which it is relevant.

  Evaluates to:

  ```
  restrictedBehaviours->forall(b : ServiceBehaviourDefinition |

    b.sLAEvents()->includes(event)
    implies
    b.evidenced(event, administration)
  )
  ```

- remainingBehaviourTimes(behaviour : ::slang::ServiceBehaviourDefinition,
  evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration,
  last : ::services::Evidence[0, *] unique) : ::types::Real[0, *]

  Informal: Returns all times that the specified evidence indicates that the specified behaviour occurred, after the specified last occurrence of the behaviour.

  Evaluates to:

  ```
  let
  next = behaviour.getNextInstanceAfter(last, evidence, administration)
  in
  if next->size() = 0
  then Set(::types::Real) {}
  else Set(::types::Real) { behaviour.getBehaviourTime(next) }->union(
    remainingBehaviourTimes(behaviour, evidence, administration, next))
  endif
  ```

- behaviourTimes(evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Real[0, *] ordered

  Informal: Returns the ordered set of times that any of the behaviours associated to with this clause are indicated to have occurred by the specified set of evidence.

  Evaluates to:

```
restrictedBehaviours->collect(b : ServiceBehaviourDefinition |

  let
  first = b.getFirstInstanceOf(evidence, administration)
  in
  if first->size() = 0
  then Set(::types::Real) {}
  else Set(::types::Real) {
    b.getBehaviourTime(first) }->union(
      remainingBehaviourTimes(b, evidence, administration, first))
  endif
)->sortedBy(t : ::types::Real | t)
```

- remainingEvidenceForBehaviourBetween(behaviour : ::slang::ServiceBehaviourDefinition,
  administration : ::services::Administration,
  last : ::services::Evidence[0, *] unique,
  start : ::types::Real,
  end : ::types::Real) : ::services::Evidence[0, *] unique

  Informal: Return the set of evidence indicating the specified behaviour between the start and end dates given, following the last instance specified.

  Evaluates to:

```
let
next = behaviour.getNextInstanceAfter(last, administration.agreed,
  administration)
in
if next->size() > 0
then
  let
  time = behaviour.getBehaviourTime(next),
  rest = remainingEvidenceForBehaviourBetween(behaviour,
    administration, next, start, end)
  in
  if time >= start and time <= end
  then next->union(rest)
  else rest
  endif
else
Set(::services::Evidence) {}
endif
```

- evidenceForBehavioursBetween(administration : ::services::Administration,
  start : ::types::Real,
  end : ::types::Real) : ::services::Evidence[0, *] unique

  Informal: Return evidence contributing to behaviours deemed to have occurred between the specified start and end times, inclusive.

  Evaluates to:

```
restrictedBehaviours->collect(b : ServiceBehaviourDefinition |
```

```
  let
  first = b.getFirstInstanceOf(administration.agreed, administration)
  in
  if first->size() > 0
  then
    let
    time = b.getBehaviourTime(first),
    rest = remainingEvidenceForBehaviourBetween(b, administration,
      first, start, end)
    in
    if time >= start and time <= end
    then first->union(rest)
    else rest
    endif
  else Set(::services::Evidence) {}
  endif
)->asSet()
```

- lastBehaviourTime(evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Real

  Informal: Determine the time of the last behaviour evident in the set of evidence specified.

  Evaluates to:

```
let times = behaviourTimes(evidence, administration)
in
times->iterate(t : ::types::Real; last : ::types::Real =
  times->any(true) |

  if t > last then t else last endif
)
```

- firstBehaviourTime(evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Real

  Informal: Determine the time of the first behaviour evident in the set of evidence specified.

  Evaluates to:

```
let times = behaviourTimes(evidence, administration)
in
times->iterate(t : ::types::Real; last : ::types::Real =
  times->any(true) |

  if t < last then t else last endif
)
```

- firstMinimalViolationIndexAfter(cutoff : ::types::Real,
  times : ::types::Real[0, *] ordered,
  administration : ::services::Administration) : ::types::Integer

  Informal: Returns the index of the behaviour time (from the specified array of behaviour times) beginning a violation after the specified cutoff time, or -1 if no such index exists.

  Evaluates to:

```
let
indices = Set(::types::Integer) { 1..times->size() }
in
indices->iterate(i : ::types::Integer; first : ::types::Integer = -1 |
```

```
  if first <> -1 then first
  else
    (
    let time = times->at(i)
    in
    let window = calculateWindow(time, administration),
    max = calculateMaxOccurrences(time, administration)
    in
    let outside = i + max
    in
    if outside > times->size() then first
    else
      let
      outsideTime = times->at(outside)
      in
      if outsideTime <= time + window
      then i
      else first
      endif
    endif
    )
  endif
)
```

- firstMinimalViolationAfter(cutoff : ::types::Real,
  evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: Given a sequence of times when behaviours associated with this clause occurred, find the first minimal subsequence causing a violation. A minimal sequence of times is the smallest set such that the times occur within the window of each other (starting with the first) and exceed maxOccurrences (calculated from the first) in number.

  Evaluates to:

```
let times = behaviourTimes(evidence,
  administration)->select(
  t : ::types::Real | t > cutoff)
in
let first = firstMinimalViolationIndexAfter(cutoff, times,
  administration)
in
if first = -1 then Set(::services::Evidence) {}
else
  let
  time = times->at(first)
  in
  let window = calculateWindow(time, administration),
  max = calculateMaxOccurrences(time, administration)
  in
  let outside = first + max
  in
  let
  outsideTime = times->at(outside)
  in
  evidenceForBehavioursBetween(administration, time, outsideTime)
endif
```

- firstMinimalViolation(evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::services::Evidence[0, *] unique

Informal: Determine the subset of the specified evidence representing the first minimal violation of this clause.

Evaluates to:

```
firstMinimalViolationAfter(-1.0, evidence, administration)
```

- nextMinimalViolation(prior : ::services::Evidence[0, *] unique,
evidence : ::services::Evidence[0, *] unique,
administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: Find the next violation subsequence given a prior subsequence and a set of times.

  Evaluates to:

```
let last = lastBehaviourTime(prior, administration)
in
firstMinimalViolationAfter(last, evidence, administration)
```

- firstMaximalViolationAfter(cutoff : ::types::Real,
evidence : ::services::Evidence[0, *] unique,
administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: Given a sequence of times when behaviours associated with this clause occurred, find the first minimal subsequence causing a violation.

  Evaluates to:

```
let times = behaviourTimes(
  evidence, administration)->select(
  t : ::types::Real | t > cutoff)
in
let indices = Sequence(::types::Integer) { 1..times->size() }
in
let first = firstMinimalViolationIndexAfter(cutoff, times,
  administration)
in
if first = -1 then Set(::services::Evidence) {}
else
  let lastFirst =
    indices->iterate(o : ::types::Integer;
      last : ::types::Integer = first |

      if o = last + 1 and firstMinimalViolationIndexAfter(
        times->at(last), times, administration) = o
      then o
      else last
      endif
    )
  in
  let outside = lastFirst +
    calculateMaxOccurrences(times->at(lastFirst),
      administration)
  in
  evidenceForBehavioursBetween(administration, times->at(first),
    times->at(outside))
endif
```

- firstMaximalViolation(evidence : ::services::Evidence[0, *] unique,
administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: Given a sequence of times when behaviours associated with this clause occurred, find the first minimal subsequence causing a violation.

  Evaluates to:

```
firstMaximalViolationAfter(-1.0, evidence, administration)
```

- nextMaximalViolation(prior : ::services::Evidence[0, *] unique,
  evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: Find the next violation subsequence given a prior subsequence and a set of times.

  Evaluates to:

```
let last = lastBehaviourTime(prior, administration)
in
firstMaximalViolationAfter(last, evidence, administration)
```

- behaviourInterval(evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Real

  Informal: Determine the duration of behaviours in the set given.

  Evaluates to:

```
lastBehaviourTime(evidence, administration) -
  firstBehaviourTime(evidence, administration)
```

Invariants:

- Wellformedness: All restricted behaviours must be defined in relation to the same service.

```
restrictedBehaviours->collect(service)->asSet()->size() = 1
```

E.2.14 Abstract class - ::slang::ServiceDefinition

Extends: ::slang::Definition, pg. 337

Definitive: A service definition identifies a service being constrained by an SLA. Services have a client and a provider and service provision results in events in the real world.

Properties:

- provider : ::slang::PartyDefinition

  Definitive: Service definitions identify the party designated as the provider of the service.

- client : ::slang::PartyDefinition

  Definitive: Service definitions identify the party designated as the client of the service.

- behaviours : ::slang::ServiceBehaviourDefinition[0, *] unique

  Opposite: ::slang::ServiceBehaviourDefinition.service : ::slang::ServiceDefinition

  Definitive: A service may have defined behaviours.

- sLA : ::slang::SLA

  Opposite: ::slang::SLA.services : ::slang::ServiceDefinition[1, *] unique

  Definitive: A service definition is part of an SLA.

Operations:

- No operations.

Invariants:

- Wellformedness: Provider and client must be defined in the same SLA.

```
provider.sLA = sLA
and
client.sLA = sLA
```

E.2.15 Class - ::slang::SLA

Extends: ::slang::Definition, pg. 337

Definitive: SLAng is a language for expressing SLAs. Concrete SLAs otherwise conforming to this type and its related SLAng syntactic types are instances of this class if and only if the parties described in the concrete SLA also agree that the SLA is in force.

SLAng, in its current form, is an abstract, extensible language. Although this class is concrete, it has mandatory components with abstract types, for which concrete subclasses do not exist in this specification. Therefore, in order to specify full SLAs using SLAng, it is currently necessary to extend this specification.

The sources for this specification are available under an open-source licence from http://uclslang.sourceforge.net/

The sources are specified in the input format accepted by the UCL MDA tools. More information concerning this language, which is based on the standard languages EMOF and OCL (http://www.omg.org), is available online at http://uclmda.sourceforge.net/

By far the most comprehensive source of information regarding this language is James Skene's PhD thesis, a link to which will shortly be made available from http://www.cs.ucl.ac.uk/staff/j.skene

Properties:

- uRI : ::types::String

  Definitive: SLAng SLAs contain a URI referencing the artifact considered to be definitive of the SLA being described.

  Informal: This is useful if the SLA exists in multiple formats. Otherwise the SLA just refers to itself.

- parties : ::slang::PartyDefinition[2, *] unique

  Opposite: ::slang::PartyDefinition.sLA : ::slang::SLA

  Definitive: The parties referred to in the SLA. There must be at least two parties corresponding to the provider and client of at least one service.

- services : ::slang::ServiceDefinition[1, *] unique

  Opposite: ::slang::ServiceDefinition.sLA : ::slang::SLA

  Definitive: SLA terms identify the services being constrained.

  Informal: Note that when describing interactions between parties, it is sometimes necessary to constrain several services in the same SLA.

- penalties : ::slang::PenaltyDefinition[0, *] unique

  Opposite: ::slang::PenaltyDefinition.sLA : ::slang::SLA

  Definitive: An SLA may define a number of pre-agreed penalties, to be levied against parties who are responsible for violations, as described by this specification.

- administrationClauses : ::slang::AdministrationClause[0, *] unique ordered

  Opposite: ::slang::AdministrationClause.sLA : ::slang::SLA

  Definitive: An SLA includes a set of administration clauses stating under what circumstances the SLA should be administered, and what is required for administration in terms of gathering evidence, submitting evidence, and calculating violations.

- auxiliaryClauses : ::slang::AuxiliaryClause[0, *] unique ordered

  Opposite: ::slang::AuxiliaryClause.sLA : ::slang::SLA

  Definitive: Any SLA terms may contain a number of additional clauses providing additional information relevant to the SLA that is not captured in service or penalty definitions, or in the conditions.

  Informal: Some types of clauses may need to be referred to from several locations in the SLA. These can be auxiliary definitions.

- events : ::services::Event[0, *] unique

  Definitive: Any number of events may occur which a relevant to an SLA. In particular, administrations of an SLA are events relevant to the SLA. SLA events are defined in administration and auxiliary clauses.

Operations:

- No operations.

Invariants:

- Wellformedness: The administration obligation: The SLA must be administered as governed by the administration clauses (in relation to service events).

```
administrationClauses->forall(a : AdministrationClause |

  a.administered()
)
```

- Wellformedness: SLA events are defined by the presence of particular types of administration clauses.

```
events = administrationClauses.sLAEvents()->asSet()
```

E.2.16 Abstract class - ::slang::TerminatingConditionClause

Extends: ::slang::ConditionClause, pg. 336

Definitive: The violation of a terminating condition clause causes the termination of the SLA.

Informal: Terminating condition clauses are not structurally different from other condition clauses. However, ordinary administration clauses should never require administrations occurring after an administration in which a violation of a terminating condition clause has been calculated.

Properties:

- No properties.

Operations:

- service() : ::slang::ServiceDefinition

  Informal: Terminating conditions clauses don't place conditions over any service.

  Evaluates to:

```
Set(ServiceDefinition) {}->any(true)
```

Invariants:

- No invariants.

E.2.17 Abstract class - ::slang::TerminationByReportAdministrationClause

Extends: ::slang::AdministrationClause, pg. 333

Definitive: An administration clause triggered by the exchange of a termination report. This class includes the obligation to monitor and report termination reports, with a specified accuracy, and optionally to administer a penalty.

This class is abstract because the calculation of any other violations at termination must be specified.

Properties:

- No properties.

Operations:

- calculateAdministrationDeadline() : ::types::Real

  Informal: (abstract) A termination-by-report administrative clause defines a deadline for administration. The SLA must have been administered according to this clause within this period of the latest date recorded in evidence relating to the termination record being sent. The length of this deadline is determined by the specific type of termination-by-report administrative clause.

- eventRelevant(administration : ::services::Administration,
  event : ::services::Event) : ::types::Boolean

  Informal: At least termination reports are relevant. Override to consider additional types of event.

  Evaluates to:

```
event.oclIsKindOf(::services::TerminationReport)
```

- administered() : ::types::Boolean

  Informal: (abstract) Checks whether the service has been administered correctly according to this clause.

  Evaluates to:

```
sLA.events->forall(e : ::services::Event |

  e.oclIsKindOf(::services::TerminationReport)
  implies
  (
    let records = e.evidence->select(oclIsKindOf(
      ::services::ReportRecord))->collect(
      oclAsType(::services::ReportRecord))
    in
    (
      let
      start = records->iterate(r : ::services::ReportRecord;
        date : ::types::Real = records->any(true).date.inMs() |

        if r.date.inMs() > date
        then r.date.inMs()
        else date
        endif
      )
      in
      (
        administrations->exists(
          a : ::services::Administration |

          let date = a.date.inMs()
          in
          (
            date >= start
            and
            date <= start +
              calculateAdministrationDeadline()
          )
        )
      )
    )
  )
)
```

Invariants:

- No invariants.

E.2.18 Abstract class - ::slang::TerminationByReportConditionClause

Extends: ::slang::TerminatingConditionClause, pg. 349

Definitive: A condition clause that awards a penalty to a party based on its peer in a service provisioning relationship submitting a termination report.

The class is abstract because the calculation of the penalty must be specified.

Properties:

- terminationReport : ::services::TerminationReport[0, 1]

  Definitive: A termination report may or may not be submitted for the SLA of which this clause forms a part.

Operations:

- sLAEvents() : ::services::Event[0, *] unique

  Informal: The inclusion of a termination report condition rules the exchange of a termination report in as a relevant event for an SLA.

  Evaluates to:

```
if not terminationReport.oclIsUndefined() then
  Set(::services::Event) { terminationReport }
else Set(::services::Event) {}
endif
```

- evidenced(event : ::services::Event,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A termination report is adequately evidenced if a report record exists referring to it.

  Evaluates to:

```
let r = event.oclAsType(::services::TerminationReport)
in
administration.agreed->exists(e : ::services::Evidence |

  e.oclIsKindOf(::services::ReportRecord)
  and
  e.oclAsType(::services::ReportRecord).report = r
)
```

- violationsCalculated(administration : ::services::Administration) : ::types::Boolean

  Informal: (abstract) Check that administrations have correctly calculated violations associated with this clause.

  Evaluates to:

```
let
agreed = administration.agreed,
violations = administration.violations
in
agreed->select(
  oclIsKindOf(::services::ReportRecord)
  and
  oclAsType(::services::ReportRecord).report.oclIsKindOf(
    ::services::TerminationReport)
)->forall(e : ::services::Evidence |

  violations->exists(v : ::services::Violation |


    v.violator =
      sLA.parties->any(
        party = oclAsType(::services::ReportRecord
          ).report.dispatcher)
    and
    v.violatedClause = self
```

```
      and
      v.evidence = Set(::services::Evidence) { e }
      and
      v.penalty = calculatePenalty(
        e.oclAsType(::services::ReportRecord), agreed)
    )
  )
```

- calculatePenalty(terminationReportRecord : ::services::ReportRecord, agreed : ::services::Evidence[0, *] unique) : ::slang::PenaltyDefinition

  Informal: (abstract) Calculate a penalty for termination.

Invariants:

- Wellformedness: If a termination report is issued it is relevant to the sLA of which this clause forms a part.

```
(not terminationReport.oclIsUndefined())
implies
terminationReport.sLA = sLA
```

- Wellformedness: If an termination-by-report condition clause is associated with an administration then a termination report accuracy clause is also required.

```
administrationClauses->forall(a : AdministrationClause |

  a.accuracyClauses->exists(
    oclIsKindOf(PermanentFixedReportRecordingAccuracyClause))
)
```

E.3 Package - ::slang::es

Informal: The es package defines all types of objects that are used only in the syntax of SLAng electronic service SLAs. At present electronic service SLAs are the only well defined type of SLA in SLAng. However, in the future there may be more types, possibly including ISP and hosting SLAs.

E.3.1 Enumeration - ::slang::es::ParameterKind

Definitive: An enumeration type for describing the directionality of parameters.

- IN

  Definitive: In parameters are used to pass information to an electronic service.

- OUT

  Definitive: Out parameters are used to return information from an electronic service.

- IN_OUT

  Definitive: In/out parameters are used to pass and return information to and from an electronic service.

E.3.2 Abstract class - ::slang::es::AvailabilityConditionClause

Extends: ::slang::ConditionClause, pg. 336

Definitive: An availability clause assigns a penalty to a period of service unavailability defined as the interval between a bug report being exchanged between the parties in a electronic-service provisioning relationship defined in an SLA, and the exchange of a corresponding bug-fix report. These reports must cite a usage mode defined as part of the definition of the electronic-service included in the SLA.

Availability clauses are abstract because more information is required to determine violations. This information includes:

- When the availability clause applies.
- Variation of penalties according to some scheme.

Properties:

- usageMode : ::slang::es::UsageModeDefinition

  Definitive: An availability clause identifies a single usage-mode that may be reported unavailable, either by the service provider, or by the client in response to the violation of a reliability clause.

- reliabilityClauses : ::slang::ServiceBehaviourRestrictionConditionClause[0, *] unique

  Definitive: violations of these reliability clauses within the calculated deadline allow the submission of a bug report by the client of the service.

Operations:

- calculateReportingDeadline(violation : ::services::Violation) : ::types::Real

  Informal: (abstract) calculate the deadline for reporting unavailability based on a violation of one of the reliability clauses.

- considerLoneBugReports() : ::types::Boolean

  Informal: (abstract) are lone bug reports considered when calculating violations?

- calculatePenaltyForBugReport(administration : ::services::Administration,
  bugReport : ::services::ReportRecord) : ::slang::PenaltyDefinition

  Informal: (abstract) calculate penalty for a lone bug-report.

- calculatePenaltyForUnavailability(administration : ::services::Administration,
  bugReport : ::services::ReportRecord,
  bugFixReport : ::services::ReportRecord) : ::slang::PenaltyDefinition

  Informal: (abstract) calculate penalty for a pair of bug and bug-fix reports.

- sLAEvents() : ::services::Event[0, *] unique

  Informal: The usage of availability clauses renders the exchange of bug and bug-fix reports relevant to the SLA.

  Evaluates to:

  ```
  (Set(::services::Event) {}->union(usageMode.bugReports)
    )->union(usageMode.bugFixReports)->asSet()
  ```

- service() : ::slang::ServiceDefinition

  Informal: The services over which this clause places conditions are the services over which the associated usage modes apply.

  Evaluates to:

  ```
  usageMode.service
  ```

- evidenced(event : ::services::Event,
  administration : ::services::Administration) : ::types::Boolean

  Informal: (abstract) Determines whether adequate evidence exists for an event (within some set of evidence) to determine violations of these conditions.

  Evaluates to:

  ```
  event.oclIsKindOf(::services::Report)
  implies
  administration.agreed->exists(e : ::services::Evidence |

    e.oclIsKindOf(::services::ReportRecord)
    and
    e.oclAsType(::services::ReportRecord).report =
      event.oclAsType(::services::Report)
  )
  ```

- bugReports(agreed : ::services::Evidence[0, *] unique) : ::services::ReportRecord[0, *] unique

  Informal: Evaluates to the set of bug reports detailed by an account of service behaviour.

  Evaluates to:

```
agreed->select(e : ::services::Evidence |

  e.oclIsKindOf(::services::ReportRecord)
)->collect(e : ::services::Evidence |

  e.oclAsType(::services::ReportRecord)
)->asSet()->select(r : ::services::ReportRecord |

  r.report.oclIsKindOf(::services::es::BugReport)
  and
  r.report.oclAsType(::services::es::BugReport).usageMode = usageMode
)
```

- findRecordOfBugFix(evidence : ::services::Evidence[0, *] unique,
  bugReport : ::services::es::BugReport) : ::services::ReportRecord

  Informal: Evaluates to the set of bug reports detailed by an account of service behaviour.

  Evaluates to:

```
evidence->any(e : ::services::Evidence |

  e.oclIsKindOf(::services::ReportRecord)
  and
  (
    let
    reportRecord = e.oclAsType(::services::ReportRecord)
    in
    reportRecord.report.oclIsKindOf(
      ::services::es::BugFixReport)
    and
    reportRecord.report.oclAsType(
      ::services::es::BugFixReport).bugReport = bugReport
  )
).oclAsType(::services::ReportRecord)
```

- violationsCalculated(administration : ::services::Administration) : ::types::Boolean

  Informal: Violations have been calculated for an administration clause if a violation exists for each pair of bug and bug-fix report, with the relevant

  Evaluates to:

```
let
evidence = administration.agreed,
violations = administration.violations
in
bugReports(evidence)->forAll(b : ::services::ReportRecord |

  let f = findRecordOfBugFix(evidence,
    b.report.oclAsType(::services::es::BugReport))
  in
  (not f.oclIsUndefined())
  implies
  violations->exists(v : ::services::Violation |
```

```
    v.evidence = Set(::services::Evidence) { b, f }
    and
    v.violator = usageMode.service.provider
    and
    v.penalty = calculatePenaltyForUnavailability(
      administration, b, f)
  )
)
and
(
  considerLoneBugReports()
  implies
  bugReports(evidence)->forAll(b : ::services::ReportRecord |

    let f = findRecordOfBugFix(evidence,
      b.report.oclAsType(::services::es::BugReport))
    in
    f.oclIsUndefined()
    implies
    violations->exists(v : ::services::Violation |

      v.evidence = Set(::services::Evidence) { b }
      and
      v.violator = usageMode.service.provider
      and
      v.penalty = calculatePenaltyForBugReport(
        administration, b)
    )
  )
)
```

- endOfLastUsage(evidence : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: Calculates the moment that the end of the last usage indicated by a set of evidence occurred.

  Evaluates to:

```
let firstUsage =
  evidence->any(oclIsKindOf(
    ::services::es::ServiceUsageRecord)).oclAsType(
    ::services::es::ServiceUsageRecord)
in
let
firstEnd = firstUsage.date.inMs() +
  (if firstUsage.duration.oclIsUndefined() then 0.0
  else firstUsage.duration.inMs() endif)
in
evidence->select(oclIsKindOf(::services::es::ServiceUsageRecord)
  )->iterate(e : ::services::Evidence; latest = firstEnd |

  let nextUsage = e.oclAsType(
    ::services::es::ServiceUsageRecord)
  in
  let nextEnd = nextUsage.date.inMs() +
    (if nextUsage.duration.oclIsUndefined() then 0.0
    else nextUsage.duration.inMs() endif)
  in
  if nextEnd > latest
  then nextEnd
```

```
  else latest
  endif
)
```

Invariants:

- Wellformedness: client-issued bug reports, records of which are included as evidence in administrations, the governing clauses of which refer to this availability clause, must be delivered within the reporting deadline of a violation of one of the referenced reliability clause.

```
administrationClauses.administrations->forall(
  a : ::services::Administration |

  a.evidence->forall(e : ::services::Evidence |

    e.oclIsKindOf(::services::ReportRecord)
    and
    (
      let record = e.oclAsType(::services::ReportRecord)
      in
      record.report.oclIsKindOf(::services::es::BugReport)
      and
      (

        let report =
          record.report.oclAsType(::services::es::BugReport)
        in
        report.usageMode = usageMode
        implies
        a.violations->exists(v : ::services::Violation |

          reliabilityClauses->includes(v.violatedClause)
          and
          record.date.inMs() >= endOfLastUsage(v.evidence)
          and
          record.date.inMs() < endOfLastUsage(v.evidence) +
            calculateReportingDeadline(v)
        )
      )
    )
  )
)
```

- Wellformedness: the behaviours restricted by the reliability clauses must all be failure modes that only exist in the usage mode referenced by this clause.

```
reliabilityClauses.restrictedBehaviours->forAll(
  b : ::slang::ServiceBehaviourDefinition |

  b.oclIsKindOf(FailureModeDefinition)
  and
  b.oclAsType(FailureModeDefinition).usageModes->asSet() =
    Set(UsageModeDefinition) { usageMode }
)
```

- Wellformedness: If an availability condition clause is associated with an administration then a report accuracy clause is also required.

```
administrationClauses->forall(a : ::slang::AdministrationClause |
```

```
    a.accuracyClauses->exists(
      oclIsKindOf(
        ::slang::PermanentFixedReportRecordingAccuracyClause))
)
```

E.3.3 Abstract class - ::slang::es::AvailabilityDependentElectronicService-UsageBehaviourDefinition

Extends: ::slang::es::ElectronicServiceUsageBehaviourDefinition, pg. 360

Informal: An available electronic-service usage behaviour is a behaviour that occurs when the service is available, according to some availability clauses.

Properties:

- availabilityClauses : ::slang::es::AvailabilityConditionClause[0, *] unique

  Definitive: An available-behaviour mode takes its definition of service availability from some availability clauses.

Operations:

- isUnavailable(usage : ::services::es::ServiceUsageRecord) : ::types::Boolean

  Informal: Calculates whether this usage is made in a mode that is currently unavailable according to a given SLA.

  Evaluates to:

```
usage.behaviours->exists(
  b : ElectronicServiceUsageBehaviourDefinition |


  b.oclIsKindOf(::slang::es::UsageModeDefinition)
  and
  (
    let usageMode = b.oclAsType(::slang::es::UsageModeDefinition)
    in
    availabilityClauses.usageMode->includes(usageMode)
    and
    usageMode.bugReports->exists(b : ::services::es::BugReport |

      b.date.inMs() <= usage.date.inMs()
      and
      not usageMode.bugFixReports->exists(
        f : ::services::es::BugFixReport |

        f.bugReport = b
        and
        f.date.inMs() <= usage.date.inMs()
      )
    )
  )
)
```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage is excluded from an available-behaviour mode if the service is unavailable at the time it occurred.

  Evaluates to:

```
isUnavailable(usage)
```

Invariants:

- No invariants.

E.3.4 Class - ::slang::es::ElectronicServiceClientDefinition

Extends: ::slang::Definition, pg. 337,
::slang::AuxiliaryClause, pg. 336

Definitive: Electronic-service SLAs include the definition of electronic-service clients which are physical devices or specific processes capable of accessing the interface of an electronic-service directly.

Properties:

- owner : ::slang::PartyDefinition

  Definitive: Electronic-service clients are controlled by a single party identified in the SLA.

- electronicServiceClient : ::services::es::ElectronicServiceClient

  Opposite: ::services::es::ElectronicServiceClient.definitions : ::slang::es::ElectronicServiceClientDefinition[0, *] unique

  Definitive: An electronic-service client definition identifies an electronic-service client in the real world.

Operations:

- No operations.

Invariants:

- No invariants.

E.3.5 Class - ::slang::es::ElectronicServiceDefinition

Extends: ::slang::ServiceDefinition, pg. 347

Definitive: An electronic service definition unambiguously identifies the service being provided in the service provision scenario that is being governed by a SLAng ES SLA. If reliability constraints are included in the SLA, the electronic service definition should include or refer to a description of the service from which it is possible to determine the correct functional behaviour of the operations of the service.

Informal: The degree of ambiguity in any referenced description of the service will affect the precision of the SLA with regards to reliability properties. The description should hence be as precise as is practically possible.

Properties:

- interfaces : ::slang::es::ElectronicServiceInterfaceDefinition[1, *] unique

  Definitive: These electronic service interfaces can be utilised by the electronic service clients referenced by this definition.

- clients : ::slang::es::ElectronicServiceClientDefinition[1, *] unique

  Definitive: The electronic service interfaces referenced by this definition may be accessed by these service clients.

Operations:

- No operations.

Invariants:

- Wellformedness : All electronic service clients must be controlled by the client of this service.

```
clients->forall(
  c : ElectronicServiceClientDefinition |

  c.owner.party = client.party
)
```

- Wellformedness: All electronic service interfaces must be controlled by the provider of this service.

```
interfaces->forall(
  i : ElectronicServiceInterfaceDefinition |

  i.owner.party = provider.party
)
```

- Wellformedness: All interfaces and ES clients must be defined in the same SLA terms as the service definition.

```
interfaces->forall(
  i : ElectronicServiceInterfaceDefinition |

  i.sLA = sLA
)
and
clients->forall(
  c : ElectronicServiceClientDefinition |

  c.sLA = sLA
)
```

E.3.6 Class - ::slang::es::ElectronicServiceInterfaceDefinition

Extends: ::slang::Definition, pg. 337,
::slang::AuxiliaryClause, pg. 336

Definitive: An electronic-service SLA includes definitions identifying electronic-service interfaces in the real world.

Properties:

- owner : ::slang::PartyDefinition

  Definitive: Electronic-service interfaces are controlled by a single party identified in the SLA.

- operations : ::slang::es::OperationDefinition[1, *] unique

  Opposite: ::slang::es::OperationDefinition.interface : ::slang::es::ElectronicServiceInterfaceDefinition

  Definitive: Electronic-service interfaces expose a set of operations defined in the SLA.

- electronicServiceInterface : ::services::es::ElectronicServiceInterface

  Opposite: ::services::es::ElectronicServiceInterface.definitions : ::slang::es::ElectronicServiceInterfaceDefinition[0, *] unique

  Definitive: An electronic-service interface definition identifies an electronic-service interface in the real world.

Operations:

- No operations.

Invariants:

- Wellformedness: The owner of the electronic service in the real world should be the same party as defined by the clause identifying the owner of the interface in the SLA.

```
owner.party = electronicServiceInterface.owner
```

E.3.7 Abstract class - ::slang::es::ElectronicServiceUsageBehaviourDefinition

Extends: ::slang::ServiceBehaviourDefinition, pg. 341

Definitive: Defines a class of failure. This can be failed or overdue responses from operations, the service as a whole, or an operation that fails repeatedly with a particular combination of parameters. It can also be failure to access up-to-date information.

Informal: Failure modes define the type of failure that reliability clauses constrain. They may relate to the functional behaviour or protocols that the interface should implement.

Properties:

- operations : ::slang::es::OperationDefinition[1, *] unique

  Definitive: The operations of the service that may cause this usage.

Operations:

- included(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: (abstract) A service usage should reference an electronic service behaviour if it is included in the behaviour (according to some administration), and not also excluded.

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: (abstract) A service usage should reference an electronic service behaviour if it is included in the behaviour (according to some administration), and not also excluded.

- sLAEvents() : ::services::Event[0, *] unique

  Informal: All events associated with the service are relevant to the determination of failures.

  Evaluates to:

```
let
electronicService = service.oclAsType(ElectronicServiceDefinition)
in
let requests =
  electronicService.interfaces.electronicServiceInterface.operations.
    requests
in
(Set(::services::Event) {}->union(requests))->union(
  requests->select(not response.oclIsUndefined()).response
  )->asSet()
```

- evidenced(event : ::services::Event,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service request is adequately evidenced if there exists a corresponding service usage record. A service response is evidenced if its corresponding request is evidenced. The service usage record should also reference this service behaviour if it is included, and not excluded, according to the definitions of this behaviour.

  Note that by this definition the inclusion of an electronic-service usage behaviour definition in a service behaviour restriction conditions implies that all usages must be recorded and reported at administration.

  Evaluates to:

```
event.oclIsKindOf(::services::es::ServiceRequest)
or
event.oclIsKindOf(::services::es::ServiceResponse)
implies
(
  let r =
    if event.oclIsKindOf(::services::es::ServiceRequest) then
```

```
      event.oclAsType(::services::es::ServiceRequest)
    else event.oclAsType(::services::es::ServiceResponse).request
    endif
  in
  administration.agreed->exists(e : ::services::Evidence |

    e.oclIsKindOf(::services::es::ServiceUsageRecord)
    and
    (
      let record = e.oclAsType(::services::es::ServiceUsageRecord)
      in
      record.request = r
    )
    and
    (
      included(e.oclAsType(::services::es::ServiceUsageRecord),
        administration)
      and
      (not excluded(e.oclAsType(
        ::services::es::ServiceUsageRecord), administration))
      implies e.oclAsType(::services::es::ServiceUsageRecord
        ).behaviours->includes(self)
    )
  )
)
```

- serviceUsageRecords(evidence : ::services::Evidence[0, *] unique) : ::services::es::ServiceUsageRecord[0, *] unique

  Informal: Find the service usage records evidencing this behaviour in a set of evidence.

  Evaluates to:

```
evidence->select(oclIsKindOf(::services::es::ServiceUsageRecord)
  )->collect(oclAsType(::services::es::ServiceUsageRecord))->
  select(record : ::services::es::ServiceUsageRecord |

    record.behaviours->includes(self)
  )->asSet()
```

- getFirstInstanceOf(evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: The service usage record with the earliest date in the presented evidence.

  Evaluates to:

```
let
records = serviceUsageRecords(evidence)
in
if records->size() = 0 then Set(::services::Evidence) {}
else
  let first = records->iterate(e : ::services::es::ServiceUsageRecord;
    first : ::services::es::ServiceUsageRecord =
      records->any(true) |

    if e.date.inMs() < first.date.inMs()
    then e
    else first
    endif
  )
```

```
  in
  Set(::services::Evidence) { first }
endif
```

- getNextInstanceAfter(prior : ::services::Evidence[0, *] unique,
  evidence : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::services::Evidence[0, *] unique

  Informal: The service usage record with the earliest date after the prior occurrence.

  Evaluates to:

```
let
records = serviceUsageRecords(evidence)
in
let last = prior->any(true).oclAsType(
  ::services::es::ServiceUsageRecord)
in
if records->size() = 0 then Set(::services::Evidence) {}
else
  let next = records->iterate(e : ::services::es::ServiceUsageRecord;
    first : ::services::es::ServiceUsageRecord = records->any(true) |

    if first.date.inMs() < last.date.inMs() or first = last
    then e
    else
      if
        e.date.inMs() < first.date.inMs() and
        e.date.inMs() > last.date.inMs()
      then e
      else first
      endif
    endif
  )
  in
  if
    (not (next.date.inMs() < last.date.inMs()))
    and
    (not (next = last))
  then Set(::services::Evidence) { next }
  else Set(::services::Evidence) {}
  endif
endif
```

- getBehaviourTime(behaviour : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: The time that the failure occurred is deemed to be the time recorded for the request on the corresponding service-usage record (behaviours in this case should only be represented by a single piece of evidence).

  Evaluates to:

```
behaviour->any(true).oclAsType(
  ::services::es::ServiceUsageRecord).date.inMs()
```

Invariants:

- No invariants.

E.3.8 Abstract class - ::slang::es::FailureModeDefinition

Extends: ::slang::es::ElectronicServiceUsageBehaviourDefinition, pg. 360

Definitive: Defines a class of failure. This can be failed or overdue responses from operations, the service as a whole, or an operation that fails repeatedly with a particular combination of parameters. It can also be failure to access up-to-date information.

Informal: Failure modes define the type of failure that reliability clauses constrain. They may relate to the functional behaviour or protocols that the interface should implement.

Properties:

- usageModes : ::slang::es::UsageModeDefinition[1, *] unique

  Opposite: ::slang::es::UsageModeDefinition.failureModes : ::slang::es::FailureModeDefinition[0, *] unique

  Definitive: Failure mode definitions must identify usage modes in which they can occur.

Operations:

- calculateResponsibleParty() : ::slang::PartyDefinition

  Informal: The provider of the electronic service is always responsible for any failures.

  Evaluates to:

```
service.provider
```

Invariants:

- Wellformedness: If a service usage references this failure mode, then it also references a usage mode in which this failure mode may occur.

```
operations.usageRecords->forAll(u : ::services::es::ServiceUsageRecord |

  u.behaviours->includes(self)
  implies
  u.behaviours->exists(b : ElectronicServiceUsageBehaviourDefinition |

    usageModes->includes(b)
  )
)
```

E.3.9 Class - ::slang::es::InformalFailureModeDefinition

Extends: ::slang::es::FailureModeDefinition, pg. 363

Definitive: Provides a concrete but informal means to define failure modes. Here we state definitively that for a failure to be regarded as included in this mode, it must conform to a fair interpretation of the textual definition of the mode given in the SLA. Hence the service usage records representing instances of the failure (or culminations of the failure should the failure manifest itself over several usages) should refer to the failure mode definition.

Informal: This definitive description of informal failure modes is binding, but unfortunately can't be formalised as the particular description of the failure is not known until the SLA is written. Formal extensions of the core language, e.g. by extending the class ::slang::es::FailureModeDefinition should be preferred to the use of this class.

Properties:

- No properties.

Operations:

- included(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal failure mode if it matches the description of the mode.

  Evaluates to:

  ```
  usage.behaviours->includes(self)
  ```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal failure mode if it matches the description of the mode.

  Evaluates to:

  ```
  not usage.behaviours->includes(self)
  ```

Invariants:

- No invariants.

E.3.10 Class - ::slang::es::InformalUsageModeDefinition

Extends: ::slang::es::UsageModeDefinition, pg. 369

Definitive: Provides a concrete but informal means to define usage modes. Here we state definitively that for a usage to be regarded as included in this mode, it must conform to a fair interpretation of the textual definition of the mode given in the SLA. Hence the service usage records representing instances of the usage (or culminations of the usage should the usage manifest itself over several invocations) should refer to the usage mode definition.

Informal: This definitive description of informal usage modes is binding, but unfortunately can't be formalised as the particular description of the usage is not known until the SLA is written. Formal extensions of the core language, e.g. by extending the class ::slang::es::UsageModeDefinition should be preferred to the use of this class.

Properties:

- No properties.

Operations:

- included(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal usage mode if it matches the description of the mode.

  Evaluates to:

  ```
  usage.behaviours->includes(self)
  ```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal usage mode if it matches the description of the mode.

  Evaluates to:

  ```
  not usage.behaviours->includes(self)
  ```

Invariants:

- No invariants.

E.3.11 Abstract class - ::slang::es::LatencyFailureModeDefinition

Extends: ::slang::es::FailureModeDefinition, pg. 363

Definitive: A failure mode including all operations that fail to respond within some time limit.

Properties:

- No properties.

Operations:

- calculateMaxDuration(date : ::types::Date) : ::types::Real

  Informal: (abstract) calculate the maximum latency of operations associated with this definition.

- included(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should be included in this mode if it's duration is greater than the calculated maximum duration (and it is not otherwise excluded).

  Evaluates to:

  ```
  not usage.duration.oclIsUndefined()
  and
  usage.duration.inMs() > calculateMaxDuration(usage.date)
  ```

Invariants:

- No invariants.

E.3.12 Class - ::slang::es::OperationDefinition

Extends: ::slang::Definition, pg. 337

Definitive: An operation definition unambiguously identifies an operation of the electronic service being provided in the service provision scenario that is being governed by a SLAng ES SLA. If a functional description of the service is provided or referenced in the service description provided in the same ES SLA, then all operation definitions should reference or reproduce parts of that description pertaining to the operation being identified. This is in order that reliability clauses associated with the operation definition can be identified with a specification of their functional behaviour.

Operation definitions also define a timeout for the operation. Requests for which responses are not received within the timeout period are regarded as failures, and should have no effect on the behaviour of the service.

Informal: An operation is a part of the interface between the client and provider of the service. The client may submit requests to the operation, and in due course expect to receive a response.

Properties:

- parameters : ::slang::es::ParameterDefinition[0, *] unique

  Opposite: ::slang::es::ParameterDefinition.operation : ::slang::es::OperationDefinition

  Definitive: Operation definitions define a set of anticipated parameters.

- interface : ::slang::es::ElectronicServiceInterfaceDefinition

  Opposite: ::slang::es::ElectronicServiceInterfaceDefinition.operations : ::slang::es::OperationDefinition[1, *] unique

  Definitive: Operation definitions are a part of the definition of an electronic-service interface.

- operation : ::services::es::Operation

  Opposite: ::services::es::Operation.definitions : ::slang::es::OperationDefinition[0, *] unique

  Definitive: An operation definition describes an operation in the real world.

- usageRecords : ::services::es::ServiceUsageRecord[0, *] unique

  Opposite: ::services::es::ServiceUsageRecord.operation : ::slang::es::OperationDefinition

  Definitive: An operation definition may be associated with several usage records.

Operations:

- No operations.

Invariants:

- No invariants.

E.3.13 Class - ::slang::es::ParameterDefinition

Extends: ::slang::Definition, pg. 337

Definitive: Defines an expected parameter of an operation.

Properties:

- parameterKind : ::slang::es::ParameterKind

  Definitive: Operation parameters may be input, output or input/output.

- parameter : ::services::es::Parameter

  Opposite: ::services::es::Parameter.definitions : ::slang::es::ParameterDefinition[0, *] unique

  Definitive: Parameter definitions identify parameters for operations of electronic services in the real world.

- parameterRecords : ::services::es::ParameterRecord[0, *]

  Opposite: ::services::es::ParameterRecord.type : ::slang::es::ParameterDefinition

  Definitive: During service usage, evidence concerning the value of parameters passed or returned from operations will accumulate, associated with this definition.

- operation : ::slang::es::OperationDefinition

  Opposite: ::slang::es::OperationDefinition.parameters : ::slang::es::ParameterDefinition[0, *] unique

  Definitive: A parameter definition is part of an operation definition.

Operations:

- isValid(value : ::types::String) : ::types::Boolean

  Informal: (abstract) Determine whether a string represents a valid encoding of a value for this parameter.

  Evaluates to:

  ```
  true
  ```

Invariants:

- Wellformedness: If the parameter kind is IN, then parameter records for this definition must all be recorded as the input in a service usage record.

  ```
  parameterKind = ParameterKind."IN"
  implies
  parameterRecords->forall(p : ::services::es::ParameterRecord |

    not p.serviceUsageAsInput.oclIsUndefined()
  )
  ```

- Wellformedness: If the parameter kind is OUT, then parameter records for this definition must all be recorded as the output in a service usage record.

  ```
  parameterKind = ParameterKind.OUT
  implies
  parameterRecords->forall(p : ::services::es::ParameterRecord |

    not p.serviceUsageAsOutput.oclIsUndefined()
  )
  ```

E.3.14 Class - ::slang::es::PermanentFixedServiceUsageRecordAccuracyClause

Extends: ::slang::es::ServiceUsageRecordAccuracyClause, pg. 367

Definitive: A service-usage accuracy clause that applies continuously and requires a fixed minimum accuracy for date and duration measurements.

Informal: The presence of a clause of this kind is mandatory if throughput or reliability clauses are used in an electronic-service SLA, to ensure a minimum level of accuracy in the reporting of service-usages.

Properties:

- dateErrorMargin : ::types::Duration

  Definitive: Service-usage records measured in accordance with this clause must report the date correctly to within this margin, with the specified confidence and type I error rate (over date and duration measurements taken together).

- durationErrorMargin : ::types::Duration

  Definitive: Service-usage records measured in accordance with this clause must report the duration of service usage correctly to within this margin, with the specified confidence and type I error rate (over date and duration measurements taken together).

Operations:

- calculateDateErrorMargin(record : ::services::es::ServiceUsageRecord,
  agreed : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: the error margin for the date measurement is the fixed value specified.

  Evaluates to:

  ```
  dateErrorMargin.inMs()
  ```

- calculateDurationErrorMargin(record : ::services::es::ServiceUsageRecord,
  agreed : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: the error margin for the duration measurement is the fixed value specified.

  Evaluates to:

  ```
  durationErrorMargin.inMs()
  ```

Invariants:

- No invariants.

E.3.15 Abstract class - ::slang::es::ServiceUsageRecordAccuracyClause

Extends: ::slang::AccuracyClause, pg. 331

Definitive:
This clause is abstract because the following information must be specified.
- How the values of the error-margin parameters vary over time.

Properties:

- No properties.

Operations:

- calculateDateErrorMargin(record : ::services::es::ServiceUsageRecord,
  agreed : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: (abstract) determine the accuracy required by the parameter sLA for the date measurement.

- calculateDurationErrorMargin(record : ::services::es::ServiceUsageRecord,
  agreed : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: (abstract) determine the accuracy required by the parameter sLA for the duration measurement.

- isDateAccurate(record : ::services::es::ServiceUsageRecord,
  agreed : ::services::Evidence[0, *] unique) : ::types::Boolean

  Informal: Assesses whether the date measurement is accurate according to the parameter sLA.

  Evaluates to:

```
record.date.inMs() >= record.request.date.inMs() -
  calculateDateErrorMargin(record, agreed)
and
record.date.inMs() <= record.request.date.inMs() +
  calculateDateErrorMargin(record, agreed)
```

- isDurationAccurate(record : ::services::es::ServiceUsageRecord,
  agreed : ::services::Evidence[0, *] unique) : ::types::Boolean

  Informal: Assesses whether the duration measurement is accurate according to the parameter sLA.

  Evaluates to:

```
if not record.response.oclIsUndefined()
then
  let trueDuration = record.response.date.inMs() -
    record.request.date.inMs()
  in
  record.duration.inMs() >=
    trueDuration - calculateDurationErrorMargin(record, agreed)
  and
  record.duration.inMs() <=
    trueDuration + calculateDurationErrorMargin(record, agreed)
else
  true
endif
```

- getMeasurementCount(evidence : ::services::Evidence[0, *] unique) : ::types::Integer

  Informal: Count the number of measurements covered by this clause for a given administration.

  Evaluates to:

```
if evidence->size() = 0 then 0
else
  evidence->collect(e : ::services::Evidence |

    if
    e.oclIsKindOf(::services::es::ServiceUsageRecord)
    then 2
    else 0
    endif
  )->sum()
endif
```

- getAccurateMeasurementCount(evidence : ::services::Evidence[0, *] unique) : ::types::Integer

  Informal: Count the number of accurate measurements covered by this clause for a given administration.

  Note that in real life this can never be evaluated with certainty. However the accuracy constraint overall can be approximately monitored using a statistical hypothesis test.

  Evaluates to:

```
if evidence->size() = 0 then 0
else
```

```
  evidence->collect(e : ::services::Evidence |

    if
    e.oclIsKindOf(::services::es::ServiceUsageRecord)
    then
      let record = e.oclAsType(::services::es::ServiceUsageRecord)
      in
      Sequence(::types::Integer) {

        if isDateAccurate(record, evidence)
        then 1
        else 0
        endif,
        if isDurationAccurate(record, evidence)
        then 1
        else 0
        endif
      }->sum()
    else 0
    endif
  )->sum()
endif
```

Invariants:

- No invariants.

E.3.16 Abstract class - ::slang::es::UsageModeDefinition

Extends: ::slang::es::ElectronicServiceUsageBehaviourDefinition, pg. 360

Definitive: Defines a way in which an electronic service can be used. This can be any subset of all possible service requests, hence the class is abstract.

Properties:

- bugReports : ::services::es::BugReport[0, *] unique

  Opposite: ::services::es::BugReport.usageMode : ::slang::es::UsageModeDefinition

  Definitive: A usage mode may be referenced by bug reports.

- bugFixReports : ::services::es::BugFixReport[0, *] unique

  Opposite: ::services::es::BugFixReport.usageMode : ::slang::es::UsageModeDefinition

  Definitive: A usage mode may be referenced by a bug-fix report.

- failureModes : ::slang::es::FailureModeDefinition[0, *] unique

  Opposite: ::slang::es::FailureModeDefinition.usageModes : ::slang::es::UsageModeDefinition[1, *] unique

  Definitive: A usage mode definition may identify failure modes that service usages in the usage mode can manifest.

Operations:

- calculateResponsibleParty() : ::slang::PartyDefinition

  Informal: The party responsible for a usage is always the service client.

  Evaluates to:

  ```
  service.client
  ```

Invariants:

- Wellformedness: Pairs of bug reports and bug-fix reports related to a usage mode should be consecutive, and not overlapping.

```
bugFixReports->forAll(f : ::services::es::BugFixReport |

  (
    not bugReports->exists(b : ::services::es::BugReport |

      b <> f.bugReport
      and
      b.date.inMs() < f.date.inMs()
      and
      b.date.inMs() >= f.bugReport.date.inMs()
    )
  )
  and
  (
    not bugFixReports->exists(f2 : ::services::es::BugFixReport |

      f2 <> f
      and
      f2.date.inMs() <= f.date.inMs()
      and
      f2.date.inMs() > f.bugReport.date.inMs()
    )
  )
)
```

E.4 Package - ::services

Informal: This package contains types definitions for all of the types of things that SLAng SLAs describe and constrain.

E.4.1 Class - ::services::Account

Definitive: An account is a collection of evidence submitted by a party to an administration.

Properties:

- party : ::services::Party

  Definitive: An account is submitted by a party.

- evidence : ::services::Evidence[0, *] unique

  Definitive: An account consists of a set of evidence.

Operations:

- No operations.

Invariants:

- Wellformedness: All evidence in an account is supported by the party that submits the account.

```
evidence->forall(e : Evidence |

  e.supporters->includes(party)
)
```

E.4.2 Class - ::services::Administration

Extends: ::services::Event, pg. 372

Definitive: An administration is an event indicating the culmination of the activity of the parties performing a reconciliation of their accounts of the service provision scenario for the administration period prior to the administration, and then calculating violations based on the reconciled account.

Informal: See the corresponding obligations in ::slang::AdministrationClause.

Properties:

- participants : ::services::Party[1, *]

  Definitive: A number of parties participate in administration by submitting evidence.

- submittedEvidence : ::services::Account[1, *] unique

  Definitive: participant parties are obliged to submit evidence in accounts that they support during the process of administration.

- administrationClause : ::slang::AdministrationClause

  Opposite: ::slang::AdministrationClause.administrations : ::services::Administration[0, *] unique

  Definitive: The administration clause triggering this administration.

- agreed : ::services::Evidence[0, *] unique

  Definitive: A single set of evidence is agreed on by the parties as the basis for the calculation of violations in the administration.

  Informal: The procedure by which this account is agreed depends on the concrete type of administration, which in turn depends on the type of SLA.

- violations : ::services::Violation[0, *] unique

  Opposite: ::services::Violation.administration : ::services::Administration

  Definitive: The semantics of SLAng define a set of violations calculated on the basis of the agreed account.

Operations:

- No operations.

Invariants:

- Wellformedness: All accounts are submitted by a participant.

```
submittedEvidence->forall(a : Account |

  participants->includes(a.party)
)
```

- Wellformedness: All evidence included in the agreed account is correlated to some evidence submitted by a participant.

```
agreed->forall(e : Evidence |

  submittedEvidence.evidence->exists(correlated(e))
)
```

- Wellformedness: All violations associated with the administration are violations of conditions associated with the administration clause associated with the administration.

```
administrationClause.conditions->includesAll(
  violations.violatedClause)
```

- Wellformedness: All participants support all evidence in the agreed account.

```
agreed->forall(e : Evidence |

  e.supporters->includesAll(participants)
)
```

- Wellformedness: No participant submits multiple accounts.

```
participants->forall(p : Party |


  submittedEvidence->exists(a : Account |

    a.party = p
  )
  implies
  submittedEvidence->one(a : Account |

    a.party = p
  )
)
```

E.4.3 Abstract class - ::services::Compensation

Extends: ::services::Event, pg. 372

Definitive: Compensation is some event mitigating the harm caused to a party by a violation of an SLA clause.

Properties:

- compensated : ::services::Party

  Definitive: Compensation is rendered to some party.

- compensating : ::services::Party

  Definitive: Compensation is rendered by some party.

- violation : ::services::Violation

  Opposite: ::services::Violation.compensation : ::services::Compensation[0, 1]

  Definitive: Compensation is rendered in respect of some violation.

Operations:

- No operations.

Invariants:

- No invariants.

E.4.4 Abstract class - ::services::Event

Definitive: An event is the completion of some activity at a specific instant of time. Events may have characteristics or attributes, some, but not necessarily all of which may be made explicit in more refined types of event described in this specification. As a result of a fair valuation of these attributes, events may conform to any number of types of events described in SLAs.

Events may be witnessed by any number of parties and generate evidence of various kinds for those parties to use in the administration of SLAs.

Properties:

- date : ::types::Date

  Definitive: The instant the event occurred.

- witnesses : ::services::Party[0, *]

  Definitive: An event may be witnessed by any number of parties.

- evidence : ::services::Evidence[0, *] unique

  Opposite: ::services::Evidence.events : ::services::Event[1, *] unique

  Definitive: Events may be instrumental in the generation of pieces of evidence of various kinds.

Operations:

- No operations.

Invariants:

- No invariants.

E.4.5 Abstract class - ::services::Evidence

Definitive: Evidence is any kind of information presented by a party for the purpose of determining whether an SLA has been violated.

Properties:

- supporters : ::services::Party[1, *]

  Opposite: ::services::Party.evidence : ::services::Evidence[0, *] unique

  Definitive: Evidence may be endorsed by a single party or may be the result of agreement between several parties.

- events : ::services::Event[1, *] unique

  Opposite: ::services::Event.evidence : ::services::Evidence[0, *] unique

  Definitive: Evidence is considered to become available in reaction to the occurrance of events.

Operations:

- correlated(other : ::services::Evidence) : ::types::Boolean

  Informal: This operation determines whether another piece of evidence makes reference to the same event, or sequence of events.

  Note that constraints based on this operation are only monitorable if correlation can be determined based on the attributes of the evidence alone.

  Evaluates to:

  ```
  other.events = events
  ```

Invariants:

- No invariants.

E.4.6 Class - ::services::Party

Definitive: Parties are people, groups or organisations who can perform some role in a service provision scenario, for example being either the client or provider of a service.

Properties:

- evidence : ::services::Evidence[0, *] unique

  Opposite: ::services::Evidence.supporters : ::services::Party[1, *]

  Definitive: Parties endorse evidence relating to events occurring in a service provision scenario, pertinent to determining violations of SLAng SLAs.

Operations:

- No operations.

Invariants:

- No invariants.

E.4.7 Abstract class - ::services::Report

Extends: ::services::Event, pg. 372

Definitive: Reports are communications between parties that are not a technical part of the delivery or use of a service. The receipt of a report is regarded as an event.

Informal: It is sometimes necessary for parties to communicate in a manner that does not use the service being constrained by the SLA. For example, if the service is broken, the client may not communicate with the server, but will wish to notify the server that an error condition needs to be rectified. Also, In the electronic service scenarios covered by our ES SLAs, there is also no way for the service provider to initiate communications with the client using the service, he must wait until the client submits a request. However, the service provider needs to communicate some information to the client when an error condition has been rectified. Reports are an abstraction of these communications, and may in fact be emails, telephone calls, carrier pigeon, or any other appropriate form of communication between the parties.

Properties:

- dispatcher : ::services::Party

  Definitive: Reports are dispatched by one party to another.

- recipient : ::services::Party

  Definitive: Reports are received by one party from another.

Operations:

- No operations.

Invariants:

- Wellformedness: Dispatcher and recipient are witnesses to the exchange of a report.

```
witnesses->includes(dispatcher)
and
witnesses->includes(recipient)
```

E.4.8 Class - ::services::ReportRecord

Extends: ::services::Evidence, pg. 373

Definitive: Evidence that a report was delivered.

Properties:

- date : ::types::Date

  Definitive: A report record records the date at which the report was delivered.

- report : ::services::Report

  Definitive: Report record evidence is produced as a result of a report being exchanged.

Operations:

- No operations.

Invariants:

- Wellformedness: The event causing this evidence to be produced is the delivery of an report.

```
events->exists(e : Event |

  e.oclIsKindOf(Report)
)
```

E.4.9 Class - ::services::TerminationReport

Extends: ::services::Report, pg. 374

Definitive: A termination notice indicates a parties decision to terminate an SLA, following final administration.

Properties:

- sLA : ::slang::SLA

  Definitive: A termination report references the SLA being terminated.

Operations:

- No operations.

Invariants:

- No invariants.

E.4.10 Class - ::services::Violation

Definitive: Violations are determined to have occurred when the behaviour of a system or a party associated with an SLA is inconsistent with the conditions established in a SLAng SLA. Violations are always the fault of a specific party, and may result in penalties being levied against that party, depending on what has been agreed in the SLA.

Properties:

- violator : ::slang::PartyDefinition

  Definitive: Violations are the fault of a specific party.

- evidence : ::services::Evidence[0, *] unique

  Definitive: Violations are supported by a set of evidence that has been agreed on by the parties as being a sound basis for assessing violations, and which are the minimal sufficent set of evidence required to determine that the violation has occurred.

- violatedClause : ::slang::ConditionClause

  Definitive: Violations identify the clause in the SLA that they violated.

- penalty : ::slang::PenaltyDefinition[0, 1]

  Opposite: ::slang::PenaltyDefinition.violations : ::services::Violation[0, *]

  Definitive: Violations identify the definition of the penalty to be applied to the violator in the SLA that they violated, if any applies.

- administration : ::services::Administration

  Opposite: ::services::Administration.violations : ::services::Violation[0, *] unique

  Definitive: Violations are calculated as part of an administration.

- compensation : ::services::Compensation[0, 1]

  Opposite: ::services::Compensation.violation : ::services::Violation

  Definitive: A violation may eventually be compensated.

Operations:

- eq(v : ::services::Violation) : ::types::Boolean

  Informal: Defines a non-object equality for violations. Violations are equal if they are supported by the same or correlated evidence, and indicate the same violator and violated clause.

  Evaluates to:

```
violator = v.violator
and
correlated(v.evidence)
and
violatedClause = v.violatedClause
```

- correlated(other : ::services::Evidence[1, *] unique) : ::types::Boolean

  Informal: Calculates whether the evidence supporting this violation is corrolated with the set provided. Condition clauses should use this operation to determine whether a violation has previously been detected relating to a set of evidence supporting the conclusion that a violation has occurred.

  Evaluates to:

```
evidence->forall(e : Evidence |

  other->exists(o : Evidence | o.correlated(e))
)
and
other->forall(o : Evidence |

  evidence->exists(e : Evidence | e.correlated(o))
)
```

Invariants:

- Wellformedness: Violations should be unique according to the non-object equality defined by the eq() function for the class.

  Informal: This implies that violations are only ever levelled against a party once for any particular set of evidence. Therefore the same violation cannot be associated with multiple administrations, and in practice violations are associated with the earliest administration in which they could be detected.

```
administration.administrationClause.sLA.administrationClauses.
  administrations.violations->forAll(
  v : Violation |


  v.eq(self)
  implies
  v = self
)
```

E.5 Package - ::services::es

Informal: The ES package contains types specific to the description of an electronic service provision scenario.

E.5.1 Class - ::services::es::BugFixReport

Extends: ::services::Report, pg. 374

Definitive: A bug-fix report is submitted by the service provider to the client to indicate that a bug that was causing some kind of service unavailability has been fixed.

Properties:

- bugReport : ::services::es::BugReport

  Opposite: ::services::es::BugReport.bugFixReport : ::services::es::BugFixReport[0, 1]

  Definitive: A bug-fix report should always identify the bug-report the fault of which is being mended.

- usageMode : ::slang::es::UsageModeDefinition

  Opposite: ::slang::es::UsageModeDefinition.bugFixReports : ::services::es::BugFixReport[0, *] unique

  Definitive: The kinds of bug believed to be being fixed.

Operations:

- No operations.

Invariants:

- Wellformedness: The usage mode being fixed should be the same usage mode reported as being problematic in the bugReport.

```
usageMode = bugReport.usageMode
```

E.5.2 Class - ::services::es::BugReport

Extends: ::services::Report, pg. 374

Definitive: A bug report is a report submitted by either the client to the service provider or vice versa, indicating that a bug is making the service unavailable to some extent.

Properties:

- usageMode : ::slang::es::UsageModeDefinition

  Opposite: ::slang::es::UsageModeDefinition.bugReports : ::services::es::BugReport[0, *] unique

  Definitive: The kinds of bug believed to be being reported.

- bugFixReport : ::services::es::BugFixReport[0, 1]

  Opposite: ::services::es::BugFixReport.bugReport : ::services::es::BugReport

  Definitive: Bug reports may be subsequently matched by a bug-fix report.

Operations:

- No operations.

Invariants:

- No invariants.

E.5.3 Class - ::services::es::ElectronicServiceClient

Definitive: A service client is a piece of software capable of making use of an electronic service, via an electronic-service interface.

Properties:

- owner : ::services::Party

  Definitive: A service client will be under the control of some party.

- definitions : ::slang::es::ElectronicServiceClientDefinition[0, *] unique

  Opposite: ::slang::es::ElectronicServiceClientDefinition.electronicServiceClient : ::services::es::ElectronicServiceClient

  Definitive: A client may be described in any number of SLAs.

Operations:

- No operations.

Invariants:

- No invariants.

E.5.4 Class - ::services::es::ElectronicServiceInterface

Definitive: An electronic service interface is a point of access to a computing service delivered by one party, the provider, to another, the client, using only electronic communication under the normal operation of the service, and not requiring the client to devote their own resources to the completion of the service. For the purposes of this specification, electronic services are accessed by the client by the submission of requests to operations, which may result in responses. The only constraints on when the client may submit requests are those specified in an SLAng SLA associated with the service to which the client is party.

Properties:

- owner : ::services::Party

  Definitive: A service interface will be under the control of some party.

- operations : ::services::es::Operation[1, *] unique

  Opposite: ::services::es::Operation.interface : ::services::es::ElectronicServiceInterface

  Definitive: Electronic services expose a number of operations that may be accessed by the client.

- definitions : ::slang::es::ElectronicServiceInterfaceDefinition[0, *] unique

  Opposite: ::slang::es::ElectronicServiceInterfaceDefinition.electronicServiceInterface : ::services::es::ElectronicServiceInterface

  Definitive: An interface may be described in any number of SLAs.

Operations:

- No operations.

Invariants:

- No invariants.

E.5.5 Class - ::services::es::Operation

Definitive: An operation is part of the interface to an electronic service. Requests may be submitted to operations by a client program and in due course a response expected (although if the service is not functioning correctly a response may not be produced).

Properties:

- interface : ::services::es::ElectronicServiceInterface

  Opposite: ::services::es::ElectronicServiceInterface.operations : ::services::es::Operation[1, *] unique

  Definitive: Operations are part of electronic services.

- parameters : ::services::es::Parameter[0, *] unique

  Opposite: ::services::es::Parameter.operation : ::services::es::Operation

  Definitive: An operation can process or return a set of parameters. Not all parameters need be used by each usage of the operation.

- requests : ::services::es::ServiceRequest[0, *] unique

  Opposite: ::services::es::ServiceRequest.operation : ::services::es::Operation

  Definitive: Requests may be made to an operation.

- definitions : ::slang::es::OperationDefinition[0, *] unique

  Opposite: ::slang::es::OperationDefinition.operation : ::services::es::Operation

  Definitive: An operation may be described in any number of SLAs.

Operations:

- No operations.

Invariants:

- No invariants.

E.5.6 Class - ::services::es::Parameter

Definitive: Operations expect to process or return certain parameters.

Properties:

- operation : ::services::es::Operation

  Opposite: ::services::es::Operation.parameters : ::services::es::Parameter[0, *] unique

  Definitive: A parameter is defined on an operation.

- definitions : ::slang::es::ParameterDefinition[0, *] unique

  Opposite: ::slang::es::ParameterDefinition.parameter : ::services::es::Parameter

  Definitive: A parameter may be described in any number of SLAs.

Operations:

- No operations.

Invariants:

- No invariants.

E.5.7 Class - ::services::es::ParameterValue

Definitive: Parameter values are components of requests to electronic services that allow the client to pass data to the service. A parameter value is also a component of a response that allows the service to pass data back to the client.

Properties:

- parameter : ::services::es::Parameter

  Definitive: A parameter value is specified for a known parameter of an operation.

- value : ::types::String

  Definitive: A parameter has some value. Values are represented in this metamodel as strings because they will be passed as an electronic signal over the network. From the point of view of the service, they may have any type.

- request : ::services::es::ServiceRequest[0, 1]

  Opposite: ::services::es::ServiceRequest.parameters : ::services::es::ParameterValue[0, *] unique

  Definitive: A parameter may be a component of a request.

- response : ::services::es::ServiceResponse[0, 1]

  Opposite: ::services::es::ServiceResponse.results : ::services::es::ParameterValue[0, *]

  Definitive: A parameter may be a component of a response.

Operations:

- No operations.

Invariants:

- No invariants.

E.5.8 Class - ::services::es::ParameterRecord

Definitive: Parameter records are records of the value of a service request or response parameter, and form part of a service usage record.

Properties:

- parameterValue : ::services::es::ParameterValue

  Definitive: The parameter record is the record of a value passed as a parameter to the operation.

- type : ::slang::es::ParameterDefinition

  Definitive: Parameter records record the value of a parameter in a manner compatible with the description of that parameter in an SLA.

- value : ::types::String

  Definitive: The parameter value rendered as a string.

- serviceUsageAsInput : ::services::es::ServiceUsageRecord[0, 1]

  Opposite: ::services::es::ServiceUsageRecord.inputs : ::services::es::ParameterRecord[0, *] unique

  Definitive: A parameter record may be associated with a service-usage record as an input.

- serviceUsageAsOutput : ::services::es::ServiceUsageRecord[0, 1]

  Opposite: ::services::es::ServiceUsageRecord.outputs : ::services::es::ParameterRecord[0, *] unique

  Definitive: A parameter record may be associated with a service-usage record as an output.

Operations:

- No operations.

Invariants:

- Wellformedness: The parameter defined by the type referenced by this record must be the same parameter for which the value is submitted.

  ```
  type.parameter = parameterValue.parameter
  ```

- Wellformedness: Parameter records are always either an input or an output in a record of a service usage.

  ```
  (
    serviceUsageAsInput.oclIsUndefined()
    or
    serviceUsageAsOutput.oclIsUndefined()
  )
  and not
  (
    serviceUsageAsInput.oclIsUndefined()
    and
    serviceUsageAsOutput.oclIsUndefined()
  )
  ```

- Wellformedness: The encoding of the parameter value as a string must be valid according to the SLA.

  ```
  type.isValid(value)
  ```

E.5.9 Class - ::services::es::ServiceRequest

Extends: ::services::Event, pg. 372

Definitive: A service request is an event in which a service client submits a request to the service across the service interface.

Informal: This is a real-world event, described in this specification for the purpose of explicating the responsibilities of the parties for service monitoring. The wellformedness invariants in ::services::es::ServiceUsageRecord describe the relationship that reported monitoring data should bear to service requests.

Properties:

- client : ::services::es::ElectronicServiceClient

  Definitive: A service client program submits the request.

- response : ::services::es::ServiceResponse[0, 1]

  Opposite: ::services::es::ServiceResponse.request : ::services::es::ServiceRequest

  Definitive: A request may result in a response being returned by the service.

- parameters : ::services::es::ParameterValue[0, *] unique

  Opposite: ::services::es::ParameterValue.request : ::services::es::ServiceRequest[0, 1]

  Definitive: A component of a request is a set of parameters that allow the client to pass data to the service.

- operation : ::services::es::Operation

  Opposite: ::services::es::Operation.requests : ::services::es::ServiceRequest[0, *] unique

  Definitive: A request is submitted to a specific operation on an electronic service interface.

Operations:

- No operations.

Invariants:

- Wellformedness: Witnesses to this event include the owner of the service client and the owner of the service interface bearing the operation being requested.

```
witnesses->includes(client.owner)
and
witnesses->includes(operation.interface.owner)
```

E.5.10 Class - ::services::es::ServiceResponse

Extends: ::services::Event, pg. 372

Definitive: A response is a message sent to the client following some processing, which in turn will have been initiated by a request submitted by the client. If the service completed successfully then the service response may return some data to the client in the form of a parameter. However, responses may also indicate an error condition.

Informal: This is a real-world event, described in this specification for the purpose of explicating the responsibilities of the parties for service monitoring. The wellformedness invariants in ::services::es::ServiceUsageRecord describe the relationship that reported monitoring data should bear to service responses.

Properties:

- request : ::services::es::ServiceRequest

  Opposite: ::services::es::ServiceRequest.response : ::services::es::ServiceResponse[0, 1]

  Definitive: Service responses occur in response to service requests.

- results : ::services::es::ParameterValue[0, *]

  Opposite: ::services::es::ParameterValue.response : ::services::es::ServiceResponse[0, 1]

  Definitive: A service response may return some data to the client.

Operations:

- No operations.

Invariants:

- Wellformedness: Witnesses to this event include the owner of the service client and the owner of the service interface bearing the operation being requested.

```
witnesses->includes(request.client.owner)
and
witnesses->includes(request.operation.interface.owner)
```

E.5.11 Class - ::services::es::ServiceUsageRecord

Extends: ::services::Evidence, pg. 373

Definitive: A service usage record is a piece of evidence concerning the use of a service. An episode of service usage incorporates a request, possibly some processing, and possibly a response. A service usage record records when the request was submitted (subject to the error characteristics of the monitor responsible for creating the record), records a possibly subjective outcome for the episode, and if a response is returned, a duration for the episode, from the request being issued to the response being returned, again subject to error.

Informal: As a piece of evidence, a service usage record is ultimately related to a particular SLA. The subjective judgements made concerning the outcome of the usage are related to the definitions included in that SLA.

Properties:

- date : ::types::Date

  Definitive: Records the time that the request is deemed to have bee submitted.

- duration : ::types::Duration[0, 1]

  Definitive: Records the amount of time between the request and the response.

- operation : ::slang::es::OperationDefinition

  Opposite: ::slang::es::OperationDefinition.usageRecords : ::services::es::ServiceUsageRecord[0, *] unique

  Definitive: Identifies the operation that was invoked in this usage.

- inputs : ::services::es::ParameterRecord[0, *] unique

  Opposite: ::services::es::ParameterRecord.serviceUsageAsInput : ::services::es::ServiceUsageRecord[0, 1]

  Definitive: Service usage records record the parameters that were passed in the request.

- outputs : ::services::es::ParameterRecord[0, *] unique

  Opposite: ::services::es::ParameterRecord.serviceUsageAsOutput : ::services::es::ServiceUsageRecord[0, 1]

  Definitive: If a result is returned by the service response it is recorded as part of the service usage record.

  Informal: No violation calculation relies on having a record of the response. However, it can be useful during reconciliation to identify failures relative to the expected behaviour of the service.

- behaviours : ::slang::es::ElectronicServiceUsageBehaviourDefinition[0, *] unique

  Definitive: Service usages may be components of a number of different types of service behaviour. The behaviours with which they are associated should be recorded in a service usage record to assist in the administration of the SLA.

  Informal: This is particularly important for administering SLAs that contain informal descriptions of behaviour, as usages constituting this behaviour must be identified at administration time.

- request : ::services::es::ServiceRequest

  Definitive: Service usage records are the result of recording a usage of the service, which starts with a request.

- response : ::services::es::ServiceResponse[0, 1]

  Definitive: Service usage records are the result of recording a usage of the service, which may include a response.

Operations:

- No operations.

Invariants:

- Wellformedness: The operation definition referenced should define the operation upon which the request was made.

```
operation.operation = request.operation
```

- Wellformedness: If a response occurs, it should be associated with this evidence, with the recorded request, and a duration should be recorded.

```
let r = events->any(oclIsKindOf(ServiceRequest)).oclAsType(
  ServiceRequest).response

in
(not r.oclIsUndefined())
implies
(
  response = r
  and
  (not duration.oclIsUndefined())
)
```

- Wellformedness: The event causing this evidence to be produced is a service request.

```
events->exists(oclIsKindOf(ServiceRequest))
```

E.6 Package - ::combined

Informal: The package combined contains all classes required to extend SLAng or its domain model in support of the definition of SLA 1 from the eMaterials case-study.

E.7 Package - ::combined::slang

Informal: The slang package in sla1 contains domain-independent syntactic extensions to the slang language.

E.7.1 Abstract class - ::combined::slang::ConsecutiveAdministrationClause

Extends: ::slang::AdministrationClause, pg. 333

Definitive: An interval administration clause finds relevant all events that it contributes to the SLA, within an interval, the administrative period.

Informal: Such administrative clauses are useful when conditions calculate sliding penalties based on events occurring in different administrative periods.

Properties:

- administrationStart : ::types::Date

  Definitive: This clause defines an earliest date, following which events are administered.

Operations:

- administrationsBetween(startDate : ::types::Real,
  endDate : ::types::Real) : ::services::Administration[0, *] unique

  Informal: Calculates the administrations of the SLA occurring between two dates.

  Evaluates to:

```
sLA.events->select(e : ::services::Event |

  e.oclIsKindOf(::services::Administration)
  and
  e.date.inMs() >= startDate
  and
  e.date.inMs() < endDate
).oclAsType(::services::Administration)->asSet()
```

- priorAdministration(date : ::types::Real) : ::services::Administration

  Informal: Returns the latest prior administration.

  Evaluates to:

```
let priors = administrationsBetween(-1.0, date)
in
priors->iterate(
  a : ::services::Administration;
  latest : ::services::Administration = priors->any(true) |

  if a.date.inMs() > latest.date.inMs()
  then a
  else latest
  endif
)
```

- intervalStartDate(administration : ::services::Administration) : ::types::Real

  Informal: (abstract) calculate the beginning of the administrative period for an administration.

  Evaluates to:

```
let prior = priorAdministration(administration.date.inMs())
in
if prior.oclIsUndefined() then administrationStart.inMs()
else prior.date.inMs()
endif
```

- intervalEndDate(administration : ::services::Administration) : ::types::Real

  Informal: (abstract) calculate the end of the administrative period for an administration.

  Evaluates to:

```
administration.date.inMs()
```

- eventRelevant(administration : ::services::Administration,
  event : ::services::Event) : ::types::Boolean

  Informal: Determines whether some event is potentially relevant to a particular administration.

  Evaluates to:

```
sLAEvents()->includes(event)
and
event.date.inMs() <= intervalEndDate(administration)
and
event.date.inMs() > intervalStartDate(administration)
```

Invariants:

- No invariants.

E.7.2 Class - ::combined::slang::FixedDeadlineFixedPoundsSterlingPaymentPenaltyDefinition

Extends: ::combined::slang::PaymentPenaltyDefinition, pg. 392

Definitive: A penalty of a fixed quantity of pounds-sterling, that must be paid within a fixed deadline.

Properties:

- amount : ::types::Real

  Definitive: This type of clause defines a fixed amount of Pounds Sterling to be paid as a penalty.

- deadline : ::types::Duration

  Definitive: This type of clause defines a fixed deadline for payments, in relation to the time of completion of the SLA administration resulting in the penalty being levied.

Operations:

- calculatePoundsSterlingPayment(violation : ::services::Violation) : ::types::Real

  Informal: Calculate the magnitude of the penalty, given the violation.

  Evaluates to:

  ```
  amount
  ```

- calculatePaymentDeadline(violation : ::services::Violation) : ::types::Real

  Informal: Calculate the payment deadline, given the violation.

  Evaluates to:

  ```
  deadline.inMs()
  ```

Invariants:

- No invariants.

E.7.3 Class - ::combined::slang::FixedPenaltyTerminationByReportConditionClause

Extends: ::slang::TerminationByReportConditionClause, pg. 350

Definitive: A condition clause that applies a fixed penalty to any party terminating the SLA by issuing a termination report.

Properties:

- fixedPenalty : ::slang::PenaltyDefinition

  Definitive: This is the fixed penalty that applies to the party issuing the termination report.

Operations:

- calculatePenalty(terminationReportRecord : ::services::ReportRecord,
  agreed : ::services::Evidence[0, *] unique) : ::slang::PenaltyDefinition

  Informal: The penalty for termination is always the fixed penalty.

  Evaluates to:

  ```
  fixedPenalty
  ```

Invariants:

- No invariants.

E.7.4 Class - ::combined::slang::PeriodicInterval

Extends: ::combined::slang::PeriodicProcess, pg. 387

Definitive: Schedules are part of the specification of when conditions in an SLA apply. The conditions specified in an SLA need not all apply at the same time. Moreover, the specification of when the conditions apply may need to be complex. Therefore all condition clauses must be associated with one or more schedules.

Informal: The effect of schedules on the determination of violations is defined by the OCL definitions contributing to the definition of violation invariants.

Each schedule expresses a number of cycles of a specified period. Within these periods, associated condition clauses first apply for a particular duration, then do not apply for the remainder of the duration. These cycles begin at a specified start date and then cease at a specified end date, which need not be a whole number of cycles later. Any clause may be associated with several schedules, and the clause applies whenever any of its schedules apply. By combining schedules in this way, complicated patterns of application can be associated with clauses.

Using schedules, it is possible to specify that several conditions clauses of the same kind apply simultaneously. Depending on the definition of violation behaviour for the clause this may result in several penalties being applied, or only the penalty from the clause that in some sense applies the most restrictive constraint.

Properties:

- duration : ::types::Duration

  Definitive: Schedules specify a duration.

  Informal: The duration of the schedule. Any clauses associated with the schedule will apply for this amount of time at the beginning of each cycle, or until the end date, whichever is sooner.

Operations:

- eq(s : ::combined::slang::PeriodicInterval) : ::types::Boolean

  Informal: Defines non-object equality for schedules. Schedules must be alike in all respects to be considered equal.

  Evaluates to:

```
duration.eq(s.duration) and
period.eq(s.period) and
startDate.eq(s.startDate) and
endDate.eq(s.endDate)
```

- applies(t : ::types::Real) : ::types::Boolean

  Informal: Evaluates to true if the schedule applies at time t, false otherwise. t is expressed in milliseconds from 00:00 1 Jan 2000 UTC+0.

  Evaluates to:

```
t >= startDate.inMs()
and
t < endDate.inMs()
and
((t - startDate.inMs()).round().mod(period.inMs().round()) <
  duration.inMs())
```

- nextDurationEndDate(t : ::types::Real) : ::types::Real

  Informal: Evaluates to the date when the next duration would end after t, if t is less than the end date of this schedule. This amounts to evaluating when the next interval of non-application of the schedule begins, assuming the duration is less than the period. t and the result are expressed in milliseconds from 00:00 1 Jan 2000 UTC+0.

  Evaluates to:

```
validateDate(

  if t < startDate.inMs() then startDate.inMs() + duration.inMs()
  else
    if applies(t)
    then
      startDate.inMs() + (cycleNumber(t) * period.inMs()) +
        duration.inMs()
    else
      nextCycleStartDate(t) + duration.inMs()
    endif
  endif
)
```

- nextEndDate(t : ::types::Real) : ::types::Real

  Informal: Returns the next date that this schedule will cease to apply after t, or -1 if it will never cease again.

  Evaluates to:

```
if duration.inMs() = period.inMs()
then
  if t < endDate.inMs()
  then
    endDate.inMs()
  else
    -1.0
  endif
else
  nextDurationEndDate(t)
endif
```

- nextStartDate(t : ::types::Real) : ::types::Real

  Informal: Returns the next date that this schedule will start to apply after t, or -1 if it will never start again.

  Evaluates to:

```
if duration = period
then
  if t < startDate.inMs()
  then
    startDate.inMs()
  else
    -1.0
  endif
else
  nextCycleStartDate(t)
endif
```

Invariants:

- No invariants.

E.7.5 Class - ::combined::slang::PeriodicProcess

Extends: ::slang::AuxiliaryClause, pg. 336

Definitive: Schedules are part of the specification of when conditions in an SLA apply. The conditions specified in an SLA need not all apply at the same time. Moreover, the specification of when the conditions apply may need to be complex. Therefore all condition clauses must be associated with one or more schedules.

Informal: The effect of schedules on the determination of violations is defined by the OCL definitions contributing to the definition of violation invariants.

Each schedule expresses a number of cycles of a specified period. Within these periods, associated condition clauses first apply for a particular duration, then do not apply for the remainder of the duration. These cycles begin at a specified start date and then cease at a specified end date, which need not be a whole number of cycles later. Any clause may be associated with several schedules, and the clause applies whenever any of its schedules apply. By combining schedules in this way, complicated patterns of application can be associated with clauses.

Using schedules, it is possible to specify that several conditions clauses of the same kind apply simultaneously. Depending on the definition of violation behaviour for the clause this may result in several penalties being applied, or only the penalty from the clause that in some sense applies the most restrictive constraint.

Properties:

- name : ::types::String

  Definitive: Schedules have names that assist in referring to them from an external context, and may provide a reminder as to the intent of the schedule.

  Informal: For example 'Every wednesday'

- startDate : ::types::Date

  Definitive: Schedules have a start date.

  Informal: Any clauses associated with the schedule will apply for the duration immediately following this date, or until the end date, whichever is sooner. The schedule will then apply again for the duration at the beginning of any subsequent cycle, or until the end date, whichever is sooner.

- period : ::types::Duration

  Definitive: Schedules specify a period.

  Informal: The period of cycles in this schedule. Any clauses associated with the schedule will apply for the duration at the beginning of each cycle, or until the end date, whichever is sooner, and then not again until this amount of time has elapsed since the beginning of the last cycle, unless associated with a different schedule that applies.

- endDate : ::types::Date

  Definitive: Schedules have an end date.

  Informal: The end date. No condition clause associated with this schedule will apply after this date, unless it is associated with a different schedule that applies.

Operations:

- eq(s : ::combined::slang::PeriodicProcess) : ::types::Boolean

  Informal: Defines non-object equality for schedules. Schedules must be alike in all respects to be considered equal.

  Evaluates to:

  ```
  period.eq(s.period) and
  startDate.eq(s.startDate) and
  endDate.eq(s.endDate)
  ```

- cycleNumber(t : ::types::Real) : ::types::Real

  Informal: Evaluates to the number of the cycle that would apply at time t, if t is after the start date and before the end date, and the cycle number is the count of cycles that have applied, starting with 0. t is expressed in milliseconds from 00:00 1 Jan 2000 UTC+0.

  Evaluates to:

  ```
  ((t - startDate.inMs()) / period.inMs()).floor()
  ```

- validateDate(t : ::types::Real) : ::types::Real

  Informal: Filters dates expressed in milliseconds from 00:00 1 Jan 2000 UTC+0. Dates outside of the start and end dates of the schedule are converted to -1, other dates remain as they are.

  Evaluates to:

```
if t < startDate.inMs() or t > endDate.inMs() then -1.0
else t
endif
```

- nextCycleStartDate(t : ::types::Real) : ::types::Real

  Informal: Evaluates to the start date of the next cycle of this schedule that would begin after t, if t is less than the end date of this schedule. t and the result are expressed in milliseconds from 00:00 1 Jan 2000 UTC+0.

  Evaluates to:

```
validateDate(

  if(t < startDate.inMs()) then startDate.inMs()
  else
    startDate.inMs() +
      ((cycleNumber(t) + 1) * period.inMs())
  endif
)
```

Invariants:

- No invariants.

E.7.6 Class - ::combined::slang::PermanentFixedWindowFixedOccurrences-FixedPenaltyMinimalServiceBehaviourRestrictionConditionClause

Extends: ::slang::ServiceBehaviourRestrictionConditionClause, pg. 341

Definitive: A service behaviour restriction clause with a fixed window size, allowing a fixed number of occurrences of the behaviour within that window.

Properties:

- maxOccurrences : ::types::Integer

  Definitive: Clauses of this type define a fixed maximum number of occurrences of the behaviours that they restrict.

- window : ::types::Duration

  Definitive: Clauses of this type define a fixed window of time within which up to the specified fixed maximum number of occurrences of the restricted behaviour may occur.

- penalty : ::slang::PenaltyDefinition

  Definitive: Clauses of this type assign a fixed penalty for minimal violations.

Operations:

- calculateMaxOccurrences(date : ::types::Real,
  administration : ::services::Administration) : ::types::Integer

  Informal: The maximum number of occurrences of the behaviour that may be observed within the sliding window starting at the time specified (in mS) is the fixed value specified in this clause.

  Evaluates to:

```
maxOccurrences
```

- calculateWindow(date : ::types::Real,
  administration : ::services::Administration) : ::types::Real

  Informal: The width of a notional sliding time window, starting at the time specified, within which no more than maxOccurances of the restricted behaviours may occur, is the fixed value specified in this clause.

  Evaluates to:

  ```
  window.inMs()
  ```

- violationExistsFor(maximal : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Check that a violation exists corresponding to a particular maximal violation, with appropriate penalty.

  Evaluates to:

  ```
  administration.violations->exists(v : ::services::Violation |

    v.evidence = maximal
    and
    v.violator = service().client
    and
    v.penalty = penalty
  )
  ```

- allLaterViolationsCalculated(prior : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Check that an administration has correctly calculated violations for all violations after some prior violation.

  Evaluates to:

  ```
  let next = nextMinimalViolation(prior, administration.agreed,
    administration)
  in
  next->size() = 0
  or
  (
    violationExistsFor(next, administration)
    and
    allLaterViolationsCalculated(next, administration)
  )
  ```

- violationsCalculated(administration : ::services::Administration) : ::types::Boolean

  Informal: Check that an administration have correctly calculated violations associated with this clause.

  Evaluates to:

  ```
  let first = firstMinimalViolation(administration.agreed, administration)
  in
  first->size() = 0
  or
  (
    violationExistsFor(first, administration)
    and
    allLaterViolationsCalculated(first, administration)
  )
  ```

Invariants:

- No invariants.

E.7.7 Abstract class - ::combined::slang::PermanentFixedWindowFixedOccurrences-MaximalServiceBehaviourRestrictionConditionClause

Extends: ::slang::ServiceBehaviourRestrictionConditionClause, pg. 341

Definitive: A service behaviour restriction clause with a fixed window size, allowing a fixed number of occurrences of the behaviour within that window.

Properties:

- maxOccurrences : ::types::Integer

  Definitive: Clauses of this type define a fixed maximum number of occurrences of the behaviours that they restrict.

- window : ::types::Duration

  Definitive: Clauses of this type define a fixed window of time within which up to the specified fixed maximum number of occurrences of the restricted behaviour may occur.

Operations:

- calculateMaxOccurrences(date : ::types::Real,
  administration : ::services::Administration) : ::types::Integer

  Informal: The maximum number of occurrences of the behaviour that may be observed within the sliding window starting at the time specified (in mS) is the fixed value specified in this clause.

  Evaluates to:

  ```
  maxOccurrences
  ```

- calculateWindow(date : ::types::Real,
  administration : ::services::Administration) : ::types::Real

  Informal: The width of a notional sliding time window, starting at the time specified, within which no more than maxOccurances of the restricted behaviours may occur, is the fixed value specified in this clause.

  Evaluates to:

  ```
  window.inMs()
  ```

- violationExistsFor(maximal : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Boolean

  Informal: (abstract) Check that a violation exists corresponding to a particular maximal violation, with appropriate penalty.

- allLaterViolationsCalculated(prior : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Check that an administration has correctly calculated violations for all violations after some prior violation.

  Evaluates to:

  ```
  let next = nextMaximalViolation(prior, administration.agreed,
    administration)
  in
  next->size() = 0
  or
  (
    violationExistsFor(next, administration)
    and
    allLaterViolationsCalculated(next, administration)
  )
  ```

- violationsCalculated(administration : ::services::Administration) : ::types::Boolean

  Informal: Check that an administration have correctly calculated violations associated with this clause.

  Evaluates to:

```
let first = firstMaximalViolation(administration.agreed, administration)
in
first->size() = 0
or
(
  violationExistsFor(first, administration)
  and
  allLaterViolationsCalculated(first, administration)
)
```

Invariants:

- No invariants.

E.7.8 Class - ::combined::slang::PermanentFixedWindowFixedOccurrencesNoPenaltyMaximalServiceBehaviourRestrictionConditionClause

Extends: ::combined::slang::PermanentFixedWindowFixedOccurrencesMaximalServiceBehaviourRestrictionConditionClause, pg. 391

Definitive: A behaviour-restriction clause with no penalty for violations.

Properties:

- No properties.

Operations:

- violationExistsFor(maximal : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Check that a violation exists corresponding to a particular maximal violation, with appropriate penalty.

  Evaluates to:

```
administration.violations->exists(v : ::services::Violation |

  v.evidence = maximal
  and
  v.violator = service().client
  and
  v.penalty.oclIsUndefined()
)
```

Invariants:

- No invariants.

E.7.9 Abstract class - ::combined::slang::PaymentPenaltyDefinition

Extends: ::slang::PenaltyDefinition, pg. 338

Definitive: A pounds-sterling payment penalty definition defines a penalty requiring a compensation payment in pounds sterling, by the violator to the injured party, within some deadline.

Properties:

- No properties.

Operations:

- calculatePoundsSterlingPayment(violation : ::services::Violation) : ::types::Real

  Informal: Calculate the magnitude of the penalty, given the violation.

- calculatePaymentDeadline(violation : ::services::Violation) : ::types::Real

  Informal: Calculate the payment deadline, given the violation.

Invariants:

- Definitive: The compensation associated with all violations to which this penalty is defined must be a pounds sterling payment, by the violator to the other party, in the amount calculated, occuring within the payment deadline

```
violations->forall(v : ::services::Violation |

  v.compensation.oclIsKindOf(
    ::combined::services::PoundsSterlingPenaltyPayment)
  and
  (
    let payment = v.compensation.oclAsType(
      ::combined::services::PoundsSterlingPenaltyPayment)
    in
    payment.date.inMs() >= v.administration.date.inMs()
    and
    payment.date.inMs() <= v.administration.date.inMs() +
      calculatePaymentDeadline(v)
    and
    payment.amount = calculatePoundsSterlingPayment(v)
    and
    payment.compensating = v.violator.party
    and
    payment.compensated =
      Set(::services::Party) {
      v.violatedClause.service().provider.party,
      v.violatedClause.service().client.party }->excluding(
        v.violator.party)->any(true)
  )
)
```

E.7.10 Class - ::combined::slang::ScheduledAdministrationClause

Extends: ::combined::slang::ScheduledClause, pg. 394,
::slang::AdministrationClause, pg. 333

Definitive: A scheduled consecutive administration clause must be administered at least once in each interval of a periodic interval. Evidence submitted must be all evidence pertinent to events occuring since the last administration, or, if this is the first event, all pertinent events occurring prior to the administration.

Properties:

- No properties.

Operations:

- priorAdministrations(date : ::types::Real) : ::services::Administration[0, *] unique

  Informal: Determine the set of administrations occurring prior to some date.

  Evaluates to:

```
sLA.events->select(e : ::services::Event |

  e.oclIsKindOf(::services::Administration)
  and
  e.date.inMs() < date
).oclAsType(::services::Administration)->asSet()
```

- administered() : ::types::Boolean

  Informal: Checks whether a set of events includes the correct administration of this clause.

  Evaluates to:

```
startDates()->forall(startDate : ::types::Real |

  priorAdministrations(startDate)->
    exists(violations->exists(violatedClause.oclIsKindOf(
      ::slang::TerminatingConditionClause)))
  or
  (
    let endDate = endDate(startDate)
    in
    let
    administrations = priorAdministrations(endDate)
    in
    administrations->one(a : ::services::Administration |

      a.date.inMs() >= startDate
      and
      a.administrationClause = self
    )
  )
)
```

Invariants:

- No invariants.

E.7.11 Abstract class - ::combined::slang::ScheduledClause

Definitive: A mixin class for clauses with a schedule influencing when penalties are assigned or administrations occur.

Properties:

- schedule : ::combined::slang::PeriodicInterval[1, *] unique ordered

Operations:

- applies(t : ::types::Real) : ::types::Boolean

  Informal: Evaluates to true if this clause applies at time t. t is expressed in milliseconds from 00:00 1 Jan 2000 UTC+0

  Evaluates to:

```
schedule->exists(i : PeriodicInterval | i.applies(t))
```

- nextStartDate(t : ::types::Real) : ::types::Real

  Informal: Returns the date (from 00:00 1 Jan 2000 TAI+0 in mS) that this clause will next start to apply after time t.

  Evaluates to:

```
schedule->iterate(i : PeriodicInterval ;
          next : PeriodicInterval = schedule->any(true) |

          if next.nextStartDate(t) = -1
          then i
          else
              if i.nextStartDate(t) = -1 then next
              else
          if i.nextStartDate(t) < next.nextStartDate(t) then i
          else next
          endif
              endif
          endif
        ).nextStartDate(t)
```

- endDate(t : ::types::Real) : ::types::Real

  Informal: Returns the next time that this clause will cease to apply after t.

  Evaluates to:

```
schedule->iterate(i : PeriodicInterval ;
    next : PeriodicInterval = schedule->any(true) |

        if next.nextEndDate(t) = -1
        then i
        else
            if i.nextEndDate(t) = -1 then next
            else
              if i.nextEndDate(t) > next.nextEndDate(t) then i
              else next
              endif
            endif
        endif
    ).nextEndDate(t)
```

- startDatesAfter(t : ::types::Real) : ::types::Real[0, *] unique ordered

  Informal: Evaluates to a list of all of the dates that this clause starts to apply after t.

  Evaluates to:

```
if nextStartDate(t) < 0
    then OrderedSet(::types::Real) {}
    else
          OrderedSet(::types::Real) { nextStartDate(t) }->union(

        startDatesAfter(nextStartDate(t))
          )
        endif
```

- startDates() : ::types::Real[0, *] unique ordered

  Informal: Evaluates to a list of all of the dates that this clause starts to apply.

  Evaluates to:

```
startDatesAfter(-1.0)
```

Invariants:

- No invariants.

E.8 Package - ::combined::slang::es

Informal: The package ::combined::slang::es contains syntactic extensions to the slang language specific to the domain of electronic services.

E.8.1 Class - ::combined::slang::es::ConsecutiveAvailabilityAwareAdministrationClause

Extends: ::combined::slang::ConsecutiveAdministrationClause, pg. 383

Definitive: A consecutive, administration aware administration clause includes events occurring within its consecutive interval, plus any events related to overlapping periods of unavailability.

Properties:

- No properties.

Operations:

- eventRelevant(administration : ::services::Administration,
  event : ::services::Event) : ::types::Boolean

  Informal: Determines whether some event is potentially relevant to a particular administration.

  Evaluates to:

```
sLAEvents()->includes(event)
and
(
  event.date.inMs() <= intervalEndDate(administration)
  and
  event.date.inMs() > intervalStartDate(administration)
)
or
(
  event.oclIsKindOf(::services::ReportRecord)
  and
  event.oclAsType(::services::ReportRecord).report.oclIsKindOf(
    ::services::es::BugReport)
  and
  event.date.inMs() <= intervalEndDate(administration)
  and
  event.oclAsType(::services::ReportRecord).report.oclAsType(
    ::services::es::BugReport).bugFixReport.date.inMs() >
    intervalEndDate(administration)
)
```

Invariants:

- No invariants.

E.8.2 Class - ::combined::slang::es::FixedDeadlineTerminationByReportConsecutiveAvailabilityAwareReconciliationAdministrationClause

Extends: ::combined::slang::es::ConsecutiveAvailabilityAwareAdministrationClause, pg. 396,
::slang::TerminationByReportAdministrationClause, pg. 349,
::slang::ReconciliationAdministrationClause, pg. 339

Definitive: A termination administration clause with a fixed deadline, which is sensitive to the need to gather evidence in relation to administrations.

Properties:

- deadline : ::types::Duration

  Definitive: Clauses of this type define a fixed amount of time following the exchange of a termination report within which the SLA must be finally administered.

Operations:

- calculateAdministrationDeadline() : ::types::Real

  Informal: Clauses of this type must be administered within a fixed deadline of a termination report being exchanged between the parties to the SLA.

  Evaluates to:

  ```
  deadline.inMs()
  ```

Invariants:

- No invariants.

E.8.3 Class - ::combined::slang::es::InformalSuccessModeDefinition

Extends: ::combined::slang::es::SuccessModeDefinition, pg. 398

Definitive: A success mode, membership of which is determined by the natural-language description given in its definition.

Properties:

- No properties.

Operations:

- included(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal success mode if it matches the description of the mode.

  Evaluates to:

  ```
  usage.behaviours->includes(self)
  ```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal success mode if it matches the description of the mode.

  Evaluates to:

  ```
  not usage.behaviours->includes(self)
  ```

Invariants:

- No invariants.

E.8.4 Class - ::combined::slang::es::ScheduledConsecutiveAvailabilityAware-ReconciliationAdministrationClause

Extends: ::combined::slang::ScheduledAdministrationClause, pg. 393,
::combined::slang::es::ConsecutiveAvailabilityAwareAdministrationClause, pg. 396,
::slang::ReconciliationAdministrationClause, pg. 339

Definitive: A scheduled, consecutive reconciliation administration clause that also requires the inclusion of any evidence relating to periods of unavailability overlapping the administrative period.

Properties:

- No properties.

Operations:

- No operations.

Invariants:

- No invariants.

E.8.5 Abstract class - ::combined::slang::es::SuccessModeDefinition

Extends: ::slang::es::ElectronicServiceUsageBehaviourDefinition, pg. 360

Definitive: A success mode is an electronic-service usage behaviour corresponding to the successful completion of a service-usage.

Properties:

- incompatibleFailureModes : ::slang::es::FailureModeDefinition[0, *] unique

  Definitive: A success mode may defines a set of failure modes with which it is incompatible.

- usageModes : ::slang::es::UsageModeDefinition[1, *] unique

  Definitive: Success mode definitions must identify usage modes in which they can occur.

Operations:

- calculateResponsibleParty() : ::slang::PartyDefinition

  Informal: The provider of the electronic service is always responsible for any successes.

  Evaluates to:

  ```
  service.provider
  ```

Invariants:

- Wellformedness: If a service usage references this success mode, then it also references a usage mode in which this success mode may occur.

  ```
  operations.usageRecords->forAll(u : ::services::es::ServiceUsageRecord |


    u.behaviours->includes(self)
    implies
    u.behaviours->exists(b :
      ::slang::es::ElectronicServiceUsageBehaviourDefinition |

      usageModes->includes(b)
    )
  )
  ```

- Wellformedness: To be in a success mode, a usage must not be in any incompatible failure mode.

  ```
  operations.usageRecords->forall(u : ::services::es::ServiceUsageRecord |

    u.behaviours->includes(self)
    implies
    not u.behaviours->exists(
      b : ::slang::es::ElectronicServiceUsageBehaviourDefinition |


      b.oclIsKindOf(::slang::es::FailureModeDefinition)
      and
      not incompatibleFailureModes->includes(b)
    )
  )
  ```

E.8.6 Abstract class - ::combined::slang::es::ViolationDependentElectronic-ServiceUsageBehaviourDefinition

Extends: ::slang::es::ElectronicServiceUsageBehaviourDefinition, pg. 360

Definitive: A violation-dependent electronic-service usage behaviour definition may not be exhibited if it contributes to a violation of some other condition.

Properties:

- satisfyingConditions : ::slang::ConditionClause[0, *] unique

  Definitive: The service usage may not be contributary evidence for violations of these conditions.

Operations:

- violating(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Determines whether a usage contributes to a violation of one of the specified clauses.

  Evaluates to:

```
administration.violations->exists(v : ::services::Violation |

  satisfyingConditions->includes(v.violatedClause)
  and
  v.evidence->includes(usage)
)
```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Requests may be excluded from this mode if the service was unavailable when they were made, or the same conditions as defined by a delegated-execution dependent failure mode apply.

  Evaluates to:

```
violating(usage, administration)
```

Invariants:

- No invariants.

E.9 Package - ::combined::services

Informal: The services package in combined contains domain-independent extension to the slang domain model.

E.9.1 Class - ::combined::services::PoundsSterlingPenaltyPayment

Extends: ::services::Compensation, pg. 372

Definitive: A penalty payment made in Pounds-Sterling.

Properties:

- amount : ::types::Real

  Definitive: In a Pounds-Sterling penalty payment, the violator pays the other SLA party an amount of money in Pounds-Sterling.

Operations:

- No operations.

Invariants:

- No invariants.

E.10 Package - ::sla1

Informal: The package sla1 contains all classes required to extend SLAng or its domain model in support of the definition of SLA 1 from the eMaterials case-study.

E.11 Package - ::sla1::slang

Informal: The slang package in sla1 contains domain-independent syntactic extensions to the slang language.

E.11.1 Class - ::sla1::slang::PermanentFixedWindowFixedOccurrencesFixedPenaltyMaximalServiceBehaviourRestrictionConditionClause

Extends: ::combined::slang::PermanentFixedWindowFixedOccurrencesMaximalServiceBehaviourRestrictionConditionClause, pg. 391

Definitive: A permanent, fixed window, fixed occurrences, fixed penalty, maximal service-behaviour restriction condition clause.

Properties:

- penalty : ::slang::PenaltyDefinition

  Definitive: Clauses of this kind related a fixed penalty to maximal violations.

Operations:

- violationExistsFor(maximal : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Check that a violation exists corresponding to a particular maximal violation, with appropriate penalty.

  Evaluates to:

```
administration.violations->exists(v : ::services::Violation |

  v.evidence = maximal
  and
  v.violator = service().provider
  and
  v.penalty = penalty
)
```

Invariants:

- No invariants.

E.11.2 Class - ::sla1::slang::PermanentFixedWindowFixedOccurrencesSteppedPenaltyMaximalServiceBehaviourRestrictionConditionClause

Extends: ::combined::slang::PermanentFixedWindowFixedOccurrencesMaximalServiceBehaviourRestrictionConditionClause, pg. 391,
::sla1::slang::SteppedPenaltyClause, pg. 401

Definitive:

Properties:

- No properties.

Operations:

- calculatePenaltyForMaximalViolation(maximal : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::slang::PenaltyDefinition

  Informal: The penalty that should apply to some maximal violation is the penalty calculated on the sliding scale.

  Evaluates to:

```
getSteppedPenalty(behaviourInterval(maximal, administration))
```

Invariants:

- No invariants.

E.11.3 Class - ::sla1::slang::SteppedPenalty

Definitive: A clause that identifies a penalty based on the duration of a violation being longer than some threshold.

Properties:

- threshold : ::types::Duration

  Definitive: The violation must last longer than this threshold, and no longer than any higher threshold for the penalty to apply.

- penalty : ::slang::PenaltyDefinition

  Definitive: This penalty applies if a violation lasts longer than the threshold, but no longer than any higher threshold.

Operations:

- No operations.

Invariants:

- No invariants.

E.11.4 Abstract class - ::sla1::slang::SteppedPenaltyClause

Definitive: A clause that associates penalties with violation on a stepped scale based on the duration of the violation.

Properties:

- penalties : ::sla1::slang::SteppedPenalty[1, *] unique ordered

Operations:

- getSteppedPenalty(violationDuration : ::types::Real) : ::slang::PenaltyDefinition

  Evaluates to:

```
let
indices = Sequence(::types::Integer) { 1..penalties->size() }
in
let
highest = indices->iterate(i : ::types::Integer;
  highest : ::types::Integer = -1 |

  if penalties->at(i).threshold.inMs() < violationDuration then i
  else highest
  endif
)
in
penalties->at(highest).penalty
```

Invariants:

- Wellformedness: Penalties should be ordered by increasing duration.

```
let
indices = Sequence(::types::Integer) { 1..penalties->size() }
in
indices->forall(i : ::types::Integer |

  i < penalties->size()
  implies
  penalties->at(i).threshold.inMs() <
    penalties->at(i + 1).threshold.inMs()
)
```

E.12 Package - ::sla1::slang::es

Informal: The package ::sla1::slang::es contains syntactic extensions to the slang language specific to the domain of electronic services.

E.12.1 Abstract class - ::sla1::slang::es::AsynchronousFailureModeDefinition

Extends: ::slang::es::FailureModeDefinition, pg. 363

Definitive: A service usage is in an asynchronous latency failure mode if it is a request for results to be produced asynchronously, and is followed, after a specified maximumLatency period, by unreliability in the retrieval operations, or unavailablity in a mode in which the retrieval operations belong.

Properties:

- asynchronousReliabilityClauses : ::slang::ServiceBehaviourRestrictionConditionClause[0, *] unique

  Definitive: If one of these reliability clauses is violated, resulting in a maximal violation ending after the maximum latency period, and before the results are retrieved, then the request should be in this failure mode.

- asynchronousAvailabilityClauses : ::slang::es::AvailabilityConditionClause[0, *] unique

  Definitive: If a period of unavailability in one of these clauses occurs, before the results are retrieved and extending beyond the maximum latency period, then the request should be in this failure mode.

- requestOperation : ::sla1::slang::es::AsynchronousOperationDefinition

  Definitive: Failure-mode definitions of this type identify a request operation, triggering the asynchronous production of results

- resultsOperations : ::sla1::slang::es::AsynchronousOperationDefinition[0, *] unique

  Definitive: Failure-mode definitions of this type identify a set of results operations, each of which must be called once to fully retrieve the results.

Operations:

- calculateLatency(request : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Real

  Informal: (abstract) Calculates a latency for the production of results. Results may be available before this time, but must be available afterwards.

- calculateRetrievalDeadline(request : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Real

  Informal: (abstract) It does not matter if requests for results are subject to unreliability of unavailability if they occur after this retrieval deadline.

- resultsId(usage : ::services::es::ServiceUsageRecord) : ::services::es::ParameterRecord

  Informal: Retrieve the results ID parameter record from a results retrieval usage.

  Evaluates to:

  ```
  let definition = resultsOperations->any(
    operation = usage.operation)
  in
  usage.inputs->union(usage.outputs)->any(
    type = definition.iDParameter)
  ```

- requestId(usage : ::services::es::ServiceUsageRecord) : ::services::es::ParameterRecord

  Informal: Retrieve the results ID parameter record from a request usage.

  Evaluates to:

  ```
  usage.inputs->union(usage.outputs)->any(
    p : ::services::es::ParameterRecord |
    p.type = requestOperation.iDParameter)
  ```

- earliestUniqueRetrievals(request : ::services::es::ServiceUsageRecord, administration : ::services::Administration) : ::services::es::ServiceUsageRecord[0, *] unique

  Informal: Find the set of earliest, unique results retrievals corresponding to a request.

  Evaluates to:

```
let
id = requestId(request),
records =
  administration.agreed->select(
    oclIsKindOf(::services::es::ServiceUsageRecord)
    )->collect(oclAsType(::services::es::ServiceUsageRecord)
    )->asSet(),
resultsOps = resultsOperations.operation->asSet()
in
let allRetrievals =
  records->select(usage : ::services::es::ServiceUsageRecord |

    usage.date.inMs() > request.date.inMs()
    and
    resultsOps->includes(usage.operation)
    and
    resultsId(usage).value = id.value
    and
    (
      let definition = resultsOperations->any(
        operation = usage.operation)
      in
      usage.behaviours->includes(definition.successMode)
    )
  )
in
allRetrievals->reject(u1 : ::services::es::ServiceUsageRecord |

  allRetrievals->exists(u2 : ::services::es::ServiceUsageRecord |

    u1.operation = u2.operation
    and
    u1.date.inMs() > u2.date.inMs()
  )
)->asSet()
```

- latestEvidenceTime(usages : ::services::Evidence[0, *] unique) : ::types::Real

  Informal: Determine the time that the last usage in a set concluded.

  Evaluates to:

```
usages->iterate(e : ::services::Evidence;
  latest = -1.0 |

  let time =
    if e.oclIsKindOf(::services::es::ServiceUsageRecord)
    then
      e.oclAsType(::services::es::ServiceUsageRecord
        ).date.inMs() +
      (
      if not e.oclAsType(::services::es::ServiceUsageRecord
        ).duration.oclIsUndefined()
      then e.oclAsType(::services::es::ServiceUsageRecord
```

```
        ).duration.inMs()
      else 0.0
      endif
      )
    else
      if e.oclIsKindOf(::services::ReportRecord)
      then e.oclAsType(::services::ReportRecord).date.inMs()
      else latest
      endif
    endif
  in
  if time > latest then time else latest endif
)
```

- included(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A usage is in this mode if it successfully triggered the production of results, but then a violation of a reliability or unavailability condition hindered the retrieval of the results by occuring after the prescribed latency (at which point results should be available) and before results could either be retrieved, or the guarantee lapses with the end of the retrieval deadline.

  Evaluates to:

```
usage.operation = requestOperation.operation
and
usage.behaviours->includes(requestOperation.successMode)
and
(
  let earliest = earliestUniqueRetrievals(usage, administration)
  in
  earliest->size() = resultsOperations->size()
  and
  (
    let cutoff = usage.date.inMs() +
      calculateRetrievalDeadline(usage, administration),
      responseTime = latestEvidenceTime(earliest)
    in
    let  response =
      if responseTime < cutoff then responseTime
      else cutoff
      endif
    in
    administration.violations->exists(v : ::services::Violation |

      Set(::slang::ConditionClause) {}->union(
        asynchronousReliabilityClauses)->union(
        asynchronousAvailabilityClauses)->includes(
          v.violatedClause)
      and
      (
        let violationEnd = latestEvidenceTime(v.evidence)
        in
        violationEnd < response
        and
        violationEnd > usage.date.inMs() +
          calculateLatency(usage, administration)
      )
    )
  )
```

```
)
```

Invariants:

- Wellformedness: The only operation that may belong to this failure mode is the request operation.

```
operations = Set(::slang::es::OperationDefinition)
  { requestOperation.operation }
```

E.12.2 Class - ::sla1::slang::es::AsynchronousOperationDefinition

Extends: ::slang::Definition, pg. 337

Definitive: An asynchronous operation definition identifies an operation that may form part of a process to asynchronously produce results. Such an operation will have a parameter identifying the batch of results being produced.

Properties:

- operation : ::slang::es::OperationDefinition

  Definitive: An asynchronous operation definition references an operation defined in some interface of a service.

- iDParameter : ::slang::es::ParameterDefinition

  Definitive: An synchronous operation definition identifies the parameter of the operation serving to identify the results being either requested or retrieved.

- successMode : ::combined::slang::es::SuccessModeDefinition

  Definitive: Requests to the operation must be in this success mode to be a correct part of the asynchronous protocol - for requests, a successful usage triggers the production of results. For result retrieval operations a successful usage retrieves the results produced asynchronously.

Operations:

- No operations.

Invariants:

- No invariants.

E.12.3 Abstract class - ::sla1::slang::es::DelegatedExecutionDependentFailureModeDefinition

Extends: ::slang::es::FailureModeDefinition, pg. 363

Definitive: A delegated execution failure mode does not include any requests for which slow execution reports have been issued.

Properties:

- No properties.

Operations:

- slowExecution(serviceUsage : ::services::es::ServiceUsageRecord, administration : ::services::Administration) : ::types::Boolean

  Informal: Has a slow execution report been issued for a given service usage in an administration.

  Evaluates to:

```
administration.agreed->exists(e : ::services::Evidence |

  e.oclIsKindOf(::services::ReportRecord)
  and
  (
    let report = e.oclAsType(::services::ReportRecord).report
    in
```

```
    report.oclIsKindOf(::sla1::services::es::SlowExecutionReport)
    and
    report.oclAsType(::sla1::services::es::SlowExecutionReport).
      requestRecord = serviceUsage
  )
)
```

- sLAEvents() : ::services::Event[0, *] unique

  Informal: Events relevant to this failure mode include the exchange of slow-execution report records.

  Evaluates to:

```
let
electronicService = service.oclAsType(
  ::slang::es::ElectronicServiceDefinition),
executables = operations.oclAsType(
  DelegatedExecutionOperationDefinition).executables
in
let requests =
  electronicService.interfaces.electronicServiceInterface.operations.
    requests
in
(Set(::services::Event) {}->union(requests))->union(
  requests->select(not response.oclIsUndefined()).response
  )->union(executables.slowExecutionReports)->asSet()
```

- evidenced(event : ::services::Event,
  administration : ::services::Administration) : ::types::Boolean

  Informal: For this failure to be evidenced, any slow execution reports related to executables related to this mode must be reported. Also, service-usage records related to usages of the operations of this mode must be produced.

  Evaluates to:

```
(
  event.oclIsKindOf(::sla1::services::es::SlowExecutionReport)
  implies
  administration.agreed->exists(e : ::services::Evidence |

    e.oclIsKindOf(::services::ReportRecord)
    and
    e.oclAsType(::services::ReportRecord).report =
      event.oclAsType(::services::Report)
  )
)
and
(
  event.oclIsKindOf(::services::es::ServiceRequest)
  or
  event.oclIsKindOf(::services::es::ServiceResponse)
  implies
  (
    let r =
      if event.oclIsKindOf(::services::es::ServiceRequest) then
        event.oclAsType(::services::es::ServiceRequest)
      else event.oclAsType(::services::es::ServiceResponse).request
      endif
    in
```

```
    administration.agreed->exists(e : ::services::Evidence |

      e.oclIsKindOf(::services::es::ServiceUsageRecord)
      and
      (
        let record = e.oclAsType(::services::es::ServiceUsageRecord)
        in
        record.request = r
      )
      and
      (
        included(e.oclAsType(::services::es::ServiceUsageRecord),
          administration)
        and
        (not excluded(e.oclAsType(
          ::services::es::ServiceUsageRecord), administration))
        implies e.oclAsType(::services::es::ServiceUsageRecord
          ).behaviours->includes(self)
      )
    )
  )
)
```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal success mode if it matches the description of the mode.

  Evaluates to:

  ```
  slowExecution(usage, administration)
  ```

Invariants:

- Wellformedness: All operations associated with this failure mode must be delegated-execution operations.

  ```
  operations->forall(oclIsKindOf(DelegatedExecutionOperationDefinition))
  ```

E.12.4 Class - ::sla1::slang::es::DelegatedExecutionOperationDefinition

Extends: ::slang::es::OperationDefinition, pg. 365

Definitive: A delegated-execution operation results in the execution of some executable maintained by a party other than the service provider. The definition identifies those executables that may potentially be executed.

Properties:

- executables : ::sla1::slang::es::ExecutableDefinition[1, *] unique

  Definitive: A clause of this kind references definitions of the executables that may be executed as a result of a call to this operation.

Operations:

- No operations.

Invariants:

- Wellformedness: The executables must be maintained by a party other than the provider of the service.

  ```
  not executables.maintainer->includes(interface.owner)
  ```

E.12.5 Abstract class - ::sla1::slang::es::ExecutableDefinition

Extends: ::slang::Definition, pg. 337,
::slang::AuxiliaryClause, pg. 336

Definitive: An executable definition identifies an executable to be executed in a delegated execution service. A maintaining party is identified. The definition also represents a guarantee that the executable will always complete in under a given duration (that may be calculated based on input value representations), on a reference node of a stated speed.

Properties:

- referenceNodeSpeed : ::types::Real

  Definitive: A scalar quantity representing the speed of the reference node on which the executable is presumed to complete in the calculated duration.

- maintainer : ::slang::PartyDefinition

  Definitive: The party responsible for maintaining the executable, hence guaranteeing its timeliness.

- slowExecutionReports : ::sla1::services::es::SlowExecutionReport[0, *] unique

  Opposite: ::sla1::services::es::SlowExecutionReport.executableDefinition : ::sla1::slang::es::ExecutableDefinition

  Definitive: An executable definition may be referenced by slow execution reports.

- executable : ::sla1::services::es::Executable

  Definitive: An executable definition identifies some executable in the real world.

Operations:

- calculateMaxDuration(inputs : ::types::String[0, *] ordered) : ::types::Real

  Informal: (abstract) Calculate maximum amount of time to complete with specified input parameters.

Invariants:

- No invariants.

E.12.6 Class - ::sla1::slang::es::FixedDurationExecutableDefinition

Extends: ::sla1::slang::es::ExecutableDefinition, pg. 408

Definitive: An executable definition guaranteeing a fixed maximum execution time (on a node of the reference speed).

Properties:

- maxDuration : ::types::Duration

  Definitive: Clauses of this type define a fixed maximum duration for executions of the defined executable on a node of the specified reference speed.

Operations:

- calculateMaxDuration(inputs : ::types::String[0, *] ordered) : ::types::Real

  Informal: The maximum amount of time to complete with any input parameters is the specified maximum duration.

  Evaluates to:

  ```
  maxDuration.inMs()
  ```

Invariants:

- No invariants.

E.12.7 Class - ::sla1::slang::es::FixedLatencyAvailabilityDependentViolationDependentFailureModeDefinition

Extends: ::slang::es::LatencyFailureModeDefinition, pg. 365,
::slang::es::AvailabilityDependentElectronicServiceUsageBehaviourDefinition, pg. 357,
::combined::slang::es::ViolationDependentElectronicServiceUsageBehaviourDefinition, pg. 398

Definitive: A failure mode defining a fixed maximum duration that service usages must not exceed.

Properties:

- maxDuration : ::types::Duration

  Definitive: A failure-mode of this kind defines a fixed maximum duration that service usages must not exceed.

Operations:

- calculateMaxDuration(date : ::types::Date) : ::types::Real

  Informal: the maximum latency of operations associated with this definition is the fixed duration specified.

  Evaluates to:

  ```
  maxDuration.inMs()
  ```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Requests may be excluded from this mode if the service was unavailable when they were made, or the same conditions as defined by a delegated-execution dependent failure mode apply.

  Evaluates to:

  ```
  isUnavailable(usage)
  or
  violating(usage, administration)
  ```

Invariants:

- No invariants.

E.12.8 Class - ::sla1::slang::es::FixedLatencyFixedDeadlineDelegatedExecutionDependentAvailabilityDependentViolationDependentAsynchronousFailureModeDefinition

Extends: ::sla1::slang::es::DelegatedExecutionDependentFailureModeDefinition, pg. 405,
::sla1::slang::es::AsynchronousFailureModeDefinition, pg. 402,
::slang::es::AvailabilityDependentElectronicServiceUsageBehaviourDefinition, pg. 357,
::combined::slang::es::ViolationDependentElectronicServiceUsageBehaviourDefinition, pg. 398

Definitive: A failure mode that is dependent on delegated execution, applies to asynchronous operations, but only when the service was in an available state when the request was made.

Properties:

- latency : ::types::Duration

  Definitive: Clauses of this type define a fixed latency, after which asynchronously calculated results must be available.

- deadline : ::types::Duration

  Definitive: Clauses of this type define a fixed deadline, within which period following the latency period, results may be retrieved once, as soon as possible, or a penalty may be levied.

Operations:

- calculateLatency(request : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Real

  Informal: (abstract) Calculates a latency for the production of results. Results may be available before this time, but must be available afterwards.

  Evaluates to:

  ```
  latency.inMs()
  ```

- calculateRetrievalDeadline(request : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Real

  Informal: (abstract) It does not matter if requests for results are subject to unreliability of unavailability if they occur after this retrieval deadline.

  Evaluates to:

  ```
  deadline.inMs()
  ```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Requests may be excluded from this mode if the service was unavailable when they were made, or the same conditions as defined by a delegated-execution dependent failure mode apply.

  Evaluates to:

  ```
  slowExecution(usage, administration)
  or
  isUnavailable(usage)
  or
  violating(usage, administration)
  ```

Invariants:

- No invariants.

E.12.9 Class - ::sla1::slang::es::InformalAvailabilityDependentViolationDependentFailureModeDefinition

Extends: ::slang::es::FailureModeDefinition, pg. 363,
::slang::es::AvailabilityDependentElectronicServiceUsageBehaviourDefinition, pg. 357,
::combined::slang::es::ViolationDependentElectronicServiceUsageBehaviourDefinition, pg. 398

Definitive:

Properties:

- No properties.

Operations:

- included(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal failure mode if it matches the description of the mode.

  Evaluates to:

  ```
  usage.behaviours->includes(self)
  ```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal failure mode if it matches the description of the mode.

  Evaluates to:

```
isUnavailable(usage)
or
violating(usage, administration)
or
not usage.behaviours->includes(self)
```

Invariants:

- No invariants.

E.12.10 Class - ::sla1::slang::es::PermanentSteppedPenaltyFixedDeadline-AvailabilityConditionClause

Extends: ::slang::es::AvailabilityConditionClause, pg. 352,
::sla1::slang::SteppedPenaltyClause, pg. 401

Definitive:

Properties:

- deadline : ::types::Duration

  Definitive: Clauses of this type define a fixed deadline in which the client may report unavailability following unreliability (according to one of the referenced conditions).

Operations:

- calculateReportingDeadline(violation : ::services::Violation) : ::types::Real

  Informal: (abstract) calculate the deadline for reporting unavailability based on a violation of one of the reliability clauses.

  Evaluates to:

```
deadline.inMs()
```

- considerLoneBugReports() : ::types::Boolean

  Informal: (abstract) are lone bug reports considered when calculating violations?

  Evaluates to:

```
true
```

- calculatePenaltyForBugReport(administration : ::services::Administration,
  bugReport : ::services::ReportRecord) : ::slang::PenaltyDefinition

  Informal: The penalty for any period of unavailability during an administrative period, is the fixed penalty.

  Evaluates to:

```
let consecutive = administration.administrationClause.
  oclAsType(::combined::slang::ConsecutiveAdministrationClause)
in
let administrationStart = consecutive.intervalStartDate(
  administration),
  administrationEnd = administration.date.inMs()
in
let violationStart =
  if administrationStart > bugReport.date.inMs()
```

```
  then administrationStart
  else bugReport.date.inMs()
  endif,
  violationEnd = administrationEnd
in
getSteppedPenalty(violationEnd - violationStart)
```

- calculatePenaltyForUnavailability(administration : ::services::Administration,
  bugReport : ::services::ReportRecord,
  bugFixReport : ::services::ReportRecord) : ::slang::PenaltyDefinition

  Informal: (abstract) calculate penalty for a pair of bug and bug-fix reports.

  Evaluates to:

```
let consecutive = administration.administrationClause.
  oclAsType(
    ::combined::slang::ConsecutiveAdministrationClause)
in
let administrationStart = consecutive.intervalStartDate(administration),
  administrationEnd = administration.date.inMs()
in
let violationStart =
    if administrationStart > bugReport.date.inMs()
    then administrationStart
    else bugReport.date.inMs()
    endif,
  violationEnd =
    if administrationStart < bugFixReport.date.inMs()
    then administrationEnd
    else bugFixReport.date.inMs()
    endif
in
getSteppedPenalty(violationEnd - violationStart)
```

Invariants:

- Wellformedness: This condition may only be associated with consecutive, availability-aware administrative clauses, because of the way it calculates penalties.

```
administrationClauses->forall(
  oclIsKindOf(
    ::combined::slang::es::
      ConsecutiveAvailabilityAwareAdministrationClause))
```

E.13 Package - ::sla1::services

Informal: The services package in sla1 contains domain-independent extension to the slang domain model.

E.14 Package - ::sla1::services::es

Informal: The package ::sla1::services::es contains extension to the slang domain model specific to the domain of electronic services.

E.14.1 Class - ::sla1::services::es::DelegatedExecution

Extends: ::services::Event, pg. 372

Definitive: A delegated execution is the execution of an executable by a service provider where responsibility for the execution time of the executable is held by another party.

Properties:

- node : ::sla1::services::es::Node
  Definitive: An execution takes place on a processing node.
- serviceRequest : ::services::es::ServiceRequest
  Definitive: An execution takes place as a consequence of a service request.
- inputs : ::sla1::services::es::ExecutionParameterValue[0, *] unique ordered
  Definitive: An execution takes a number of input parameters.
- outputs : ::sla1::services::es::ExecutionParameterValue[0, *] unique
  Definitive: An execution may produce various results.
- executable : ::sla1::services::es::Executable
  Definitive: An execution is of an executable.
- duration : ::types::Duration
  Definitive: An executation takes a certain amount of time to complete.

Operations:

- No operations.

Invariants:

- No invariants.

E.14.2 Class - ::sla1::services::es::Executable

Definitive: An executable is a package of program code appropriate for execution on some processing nodes.

Properties:

- No properties.

Operations:

- No operations.

Invariants:

- No invariants.

E.14.3 Class - ::sla1::services::es::ExecutionParameterRecord

Definitive: An execution parameter record records the value of an input to, or output from an execution.

Properties:

- value : ::types::String
  Definitive: A string representation of the execution parameter value.
- executionParameterValue : ::sla1::services::es::ExecutionParameterValue
  Definitive : An execution parameter record records the value of some execution parameter.

Operations:

- No operations.

Invariants:

- Wellformedness: The parameter record should capture the parameter value identically.

```
value = executionParameterValue.value
```

E.14.4 Class - ::sla1::services::es::ExecutionParameterValue

Definitive: An execution parameter is some data taken as an input by an executable, or produced during an execution.

Properties:

- value : ::types::String

  Definitive: An execution parameter value is assumed to have a string representation.

Operations:

- No operations.

Invariants:

- No invariants.

E.14.5 Class - ::sla1::services::es::Node

Definitive: A node is a processing platform. Nodes may be of various architectural types.

Properties:

- speed : ::types::Real

  Definitive: It is assumed that within the architectural type of a node, some scalar value represents its absolute speed, such that the ratio of two values of speed for two nodes of the same architectural type will be a good estimate of the ratio of execution times for two executions of the same executable (appropriate to the architecture) on the two nodes.

Operations:

- No operations.

Invariants:

- No invariants.

E.14.6 Class - ::sla1::services::es::SlowExecutionReport

Extends: ::services::Report, pg. 374

Definitive: A slow execution report represents a complaint by a service provider to the maintainer of an executable that normalised execution time for an execution has taken longer than guaranteed by the maintainer.

Properties:

- duration : ::types::Duration

  Definitive: A slow-execution report contains a record of the normalised duration of the execution

- requestRecord : ::services::es::ServiceUsageRecord

  Definitive: A slow-execution report includes a record of the particulars of the service request that led to the execution taking place. This allows the maintainer to establish that the execution parameters chosen were legitimate.

- inputs : ::sla1::services::es::ExecutionParameterRecord[0, *] unique ordered

  Definitive: A slow-execution report includes a record of the values of any inputs to the executable.

- executableDefinition : ::sla1::slang::es::ExecutableDefinition

  Opposite: ::sla1::slang::es::ExecutableDefinition.slowExecutionReports : ::sla1::services::es::SlowExecutionReport[0, *] unique

  Definitive: A slow-execution report refers to an executable identified in an SLA.

- delegatedExecution : ::sla1::services::es::DelegatedExecution

  Definitive: A slow-execution report is a result of observing a delegated execution.

Operations:

- trueNormalisedDuration() : ::types::Real

  Informal: Calculates the true normalised duration.

  Evaluates to:

```
delegatedExecution.duration.inMs() *
  (executableDefinition.referenceNodeSpeed /
    delegatedExecution.node.speed)
```

Invariants:

- Wellformedness: The executable definition referred to by the report must identify the executable used in the execution.

```
delegatedExecution.executable = executableDefinition.executable
```

- Wellformedness: The service request referenced by the report mus refer to a delegated execution definition referencing the executable that ran too slowly.

```
requestRecord.operation.oclIsKindOf(
  ::sla1::slang::es::DelegatedExecutionOperationDefinition)
and
requestRecord.operation.oclAsType(
  ::sla1::slang::es::DelegatedExecutionOperationDefinition).
  executables->includes(executableDefinition)
```

- Wellformedness: The values measured by the input parameter records must be the input values to the execution.

```
Sequence(::types::Integer) { 1..delegatedExecution.inputs->size() }->
  forall(i : ::types::Integer |

  inputs->at(i).executionParameterValue =
    delegatedExecution.inputs->at(i)
)
```

E.15 Package - ::sla4

Informal: The package sla1 contains all classes required to extend SLAng or its domain model in support of the definition of SLA 1 from the eMaterials case-study.

E.16 Package - ::sla4::slang

Informal: The slang package in sla1 contains domain-independent syntactic extensions to the slang language.

E.16.1 Class - ::sla4::slang::FixedDeadlineScalingPoundsSterlingPayment-PenaltyDefinition

Extends: ::combined::slang::PaymentPenaltyDefinition, pg. 392

Definitive:

Properties:

- amountPerHour : ::types::Real

  Definitive: This type of clause defines a fixed amount of Pounds Sterling to be paid per hour of the violation.

- deadline : ::types::Duration

  Definitive: This type of clause defines a fixed deadline for payments, in relation to the time of completion of the SLA administration resulting in the penalty being levied.

Operations:

- calculatePoundsSterlingPayment(violation : ::services::Violation) : ::types::Real

  Informal: Calculate the magnitude of the penalty, given the violation.

  Evaluates to:

```
amountPerHour * (violation.violatedClause.oclAsType(
  ScalingPenaltyConditionClause).calculateViolationDuration(
    violation) / 3600000.0)
```

- calculatePaymentDeadline(violation : ::services::Violation) : ::types::Real

  Informal: Calculate the payment deadline, given the violation.

  Evaluates to:

```
deadline.inMs()
```

Invariants:

- No invariants.

E.16.2 Class - ::sla4::slang::PermanentFixedWindowFixedOccurrencesScaling-PenaltyMaximalServiceBehaviourRestrictionConditionClause

Extends: ::combined::slang::PermanentFixedWindowFixedOccurrencesMaximalServiceBehaviour-RestrictionConditionClause, pg. 391,
::sla4::slang::ScalingPenaltyConditionClause, pg. 417

Definitive:

Properties:

- No properties.

Operations:

- calculateViolationDuration(violation : ::services::Violation) : ::types::Real

  Informal: Scaling penalties need to be able to calculate the duration of a violation given the evidence that constitutes it.

  Evaluates to:

```
behaviourInterval(violation.evidence,
  violation.administration)
```

- violationExistsFor(maximal : ::services::Evidence[0, *] unique,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Check that a violation exists corresponding to a particular maximal violation, with appropriate penalty.

  Evaluates to:

```
administration.violations->exists(v : ::services::Violation |

  v.evidence = maximal
  and
  v.violator = service().provider
  and
  v.penalty = penalty
)
```

Invariants:

- No invariants.

E.16.3 Abstract class - ::sla4::slang::ScalingPenaltyConditionClause

Extends: ::slang::ConditionClause, pg. 336

Definitive: A scaling-penalty condition clause assigns a penalty that varies with the duration of a violation.

Properties:

- penalty : ::slang::PenaltyDefinition

  Definitive: Clauses of this type associate a fixed penalty, that may be scaling, with violations.

Operations:

- calculateViolationDuration(violation : ::services::Violation) : ::types::Real

  Informal: (abstract) Scaling penalties need to be able to calculate the duration of a violation given the evidence that constitutes it.

Invariants:

- No invariants.

E.17 Package - ::sla4::slang::es

Informal: The package ::sla4::slang::es contains syntactic extensions to the slang language specific to the domain of electronic services.

E.17.1 Class - ::sla4::slang::es::ScheduledFixedLatencyAvailabilityDependent-ViolationDependentFailureModeDefinition

Extends: ::slang::es::LatencyFailureModeDefinition, pg. 365,
::slang::es::AvailabilityDependentElectronicServiceUsageBehaviourDefinition, pg. 357,
::combined::slang::es::ViolationDependentElectronicServiceUsageBehaviourDefinition, pg. 398,
::combined::slang::ScheduledClause, pg. 394

Definitive: A failure mode defining a fixed maximum duration that service usages must not exceed.

Properties:

- maxDuration : ::types::Duration

  Definitive: A failure-mode of this kind defines a fixed maximum duration that service usages must not exceed.

Operations:

- calculateMaxDuration(date : ::types::Date) : ::types::Real

  Informal: the maximum latency of operations associated with this definition is the fixed duration specified.

  Evaluates to:

  ```
  maxDuration.inMs()
  ```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: Requests may be excluded from this mode if the service was unavailable when they were made, or the same conditions as defined by a delegated-execution dependent failure mode apply.

  Evaluates to:

  ```
  isUnavailable(usage)
  or
  violating(usage, administration)
  or
  (not applies(usage.date.inMs()))
  ```

Invariants:

- No invariants.

E.17.2 Class - ::sla4::slang::es::ScheduledInformalAvailabilityDependent-ViolationDependentFailureModeDefinition

Extends: ::slang::es::FailureModeDefinition, pg. 363,
::slang::es::AvailabilityDependentElectronicServiceUsageBehaviourDefinition, pg. 357,
::combined::slang::es::ViolationDependentElectronicServiceUsageBehaviourDefinition, pg. 398,
::combined::slang::ScheduledClause, pg. 394

Definitive: An informal, availability-dependent, violation-dependent scheduled failure mode. Usages should be identified as being in this mode if they match the informal description of this mode given, and the service is not unavailable according to some clause, the usage is not part of a violation of some other clause(s), and this clause applies according to some schedule.

Properties:

- No properties.

Operations:

- included(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal failure mode if it matches the description of the mode.

  Evaluates to:

  ```
  usage.behaviours->includes(self)
  ```

- excluded(usage : ::services::es::ServiceUsageRecord,
  administration : ::services::Administration) : ::types::Boolean

  Informal: A service usage should reference an informal failure mode if it matches the description of the mode.

  Evaluates to:

  ```
  isUnavailable(usage)
  or
  violating(usage, administration)
  or
  (not applies(usage.date.inMs()))
  or
  not usage.behaviours->includes(self)
  ```

Invariants:

- No invariants.

E.17.3 Class - ::sla4::slang::es::ScheduledScalingPenaltyFixedDeadlineAvailability-ConditionClause

Extends: ::slang::es::AvailabilityConditionClause, pg. 352,
::sla4::slang::ScalingPenaltyConditionClause, pg. 417,
::combined::slang::ScheduledClause, pg. 394

Definitive:

Properties:

- deadline : ::types::Duration

  Definitive: Clauses of this type define a fixed deadline in which the client may report unavailability following unreliability (according to one of the referenced conditions).

Operations:

- calculateReportingDeadline(violation : ::services::Violation) : ::types::Real

  Informal: (abstract) calculate the deadline for reporting unavailability based on a violation of one of the reliability clauses.

  Evaluates to:

  ```
  deadline.inMs()
  ```

- considerLoneBugReports() : ::types::Boolean

  Informal: (abstract) are lone bug reports considered when calculating violations?

  Evaluates to:

  ```
  true
  ```

- calculatePenaltyForBugReport(administration : ::services::Administration,
  bugReport : ::services::ReportRecord) : ::slang::PenaltyDefinition

  Informal: The penalty for any period of unavailability during an administrative period, is the fixed penalty.

  Evaluates to:

  ```
  penalty
  ```

- calculatePenaltyForUnavailability(administration : ::services::Administration,
  bugReport : ::services::ReportRecord,
  bugFixReport : ::services::ReportRecord) : ::slang::PenaltyDefinition

  Informal: (abstract) calculate penalty for a pair of bug and bug-fix reports.

  Evaluates to:

  ```
  penalty
  ```

- latestStartBefore(date : ::types::Real) : ::types::Real

  Informal: Determine the latest start date of this clause prior to some date.

  Evaluates to:

  ```
  let allStartDates = startDates()
  in
  allStartDates->iterate(s : ::types::Real;
    latest = allStartDates->any(self <= date) |

    if latest.oclIsUndefined()
    then latest
    else
      if s > latest and s <= date
      then s
      else latest
      endif
    endif
  )
  ```

- applicationTimeBetween(start : ::types::Real,
  end : ::types::Real) : ::types::Real

  Informal: Calculated for how many milliseconds this clause applies between the specified start and end dates.

  Evaluates to:

```
if latestStartBefore(start).oclIsUndefined()
then
  let earliestStart = nextStartDate(start)
  in
  applicationTimeBetween(earliestStart, end)
else
  let earliestEnd = endDate(latestStartBefore(start))
  in
  if earliestEnd < start
  then
    let earliestStart = nextStartDate(start)
    in
    applicationTimeBetween(earliestStart, end)
  else
    if earliestEnd < end
    then
      (earliestEnd - start) +
        applicationTimeBetween(earliestEnd, end)
    else end - start
    endif
  endif
endif
```

- calculateViolationDuration(violation : ::services::Violation) : ::types::Real

  Informal: Scaling penalties need to be able to calculate the duration of a violation given the evidence that constitutes it.

  Evaluates to:

```
let consecutive = violation.administration.administrationClause.
  oclAsType(
    ::combined::slang::ConsecutiveAdministrationClause),
  bugReport = violation.evidence->any(
    oclIsKindOf(::services::ReportRecord)
    and
    oclAsType(::services::ReportRecord).report.oclIsKindOf(
      ::services::es::BugReport)).oclAsType(
      ::services::ReportRecord),
  bugFixReport = violation.evidence->any(
    oclIsKindOf(::services::ReportRecord)
    and
    oclAsType(::services::ReportRecord).report.oclIsKindOf(
      ::services::es::BugFixReport)).oclAsType(
      ::services::ReportRecord)
in
let administrationStart = consecutive.intervalStartDate(
  violation.administration),
  administrationEnd = violation.administration.date.inMs()
in
let violationStart =
  if bugReport.oclIsUndefined()
  then administrationStart
  else
    if administrationStart > bugReport.date.inMs()
    then administrationStart
    else bugReport.date.inMs()
    endif
  endif,
  violationEnd =
```

```
  if bugFixReport.oclIsUndefined()
  then administrationEnd
  else
    if administrationStart < bugFixReport.date.inMs()
    then administrationEnd
    else bugFixReport.date.inMs()
    endif
  endif
in
applicationTimeBetween(violationStart, violationEnd)
```

Invariants:

- Wellformedness: This condition may only be associated with consecutive, availability-aware administrative clauses, because of the way it calculates penalties.

```
administrationClauses->forall(
  oclIsKindOf(
    ::combined::slang::es::
      ConsecutiveAvailabilityAwareAdministrationClause))
```

Appendix F

Bibliography

[1] The ActiveBPEL open source engine project. `http://www.active-endpoints.com/active-bpel-engine-overview.htm`.

[2] AndroMDA code generation tool. `http://www.andromda.org/`.

[3] S. Ansaloni, A. Sztajnberg, R. C. Cerqueira, and O. Loques. Deploying QoS contracts in the architectural level. In *Workshop on Architecture Description Languages (WADL04) - IFIP WCC'2004*, pages 11–20, August 2004.

[4] The Apache HTTP Server project. `http://httpd.apache.org/`.

[5] The Apache Jakarta Project. *Apache Jakarta Tomcat servlet container*. `http://jakarta.apache.org/tomcat/`.

[6] The Apache Jakarta Project. *Apache JMeter*. `http://jakarta.apache.org/jmeter/`.

[7] The Apache Axis Platform. `http://ws.apache.org/axis/`.

[8] A. Baroni and F. Abreu. Formalizing object-oriented design metrics upon the UML meta-model. In *16th Brazilian Symposium on Software Engineering, Gramado, Brazil*, 2002.

[9] H. Bauerdick, M. Gogolla, and F. Gutsche. Detecting OCL traps in the UML 2.0 superstructure: An experience report. In *UML 2004*, number 3273 in Lecture Notes in Computer Science (LNCS), pages 188–196. Springer-Verlag, 2004.

[10] C. Becker and K. Geihs. Generic QoS-support for CORBA. In *Fifth IEEE Symposium on Computers and Communications (ISCC 2000)*, page 60. IEEE Press, July 2000.

[11] C. Bettini. Obligation monitoring in policy management. In *Third International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*. IEEE Computer Society, 2002.

[12] J.G. Cederquist, R. Corin, M.A.C Dekker, S. Etalle, and J.I. den Hartog. An audit logic for accountabilty. In *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*. IEEE Computer Society, 2005.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[14] The Condor Project. http://www.cs.wisc.edu/condor/.

[15] N. Cook, S. Shrivastava, and S. Wheater. Distributed object middleware to support dependable information sharing between organisations. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2002.

[16] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Policy 2001: Workshop on Policies for Distributed Systems and Networks*, number 1995 in Lecture Notes in Computer Science, pages 18 – 39. Springer-Verlag, 2001.

[17] Digital Equipment Corporation (DEC). *Programming with ONC RPC*, 1992. http://www.cs.arizona.edu/computer.help/policy/DIGITAL_unix/AA-Q0R5B-TET1_html/TITLE.html.

[18] Distributed Management Task Force, inc. *Common Information Model (CIM) Specification*, June 1999. http://www.dmtf.org/standards/cim/.

[19] The Eclipse Project. *Eclipse*. http://www.eclipse.org/.

[20] The Eclipse Project. *The Eclipse BPEL Project*. http://www.eclipse.org/bpel/index.php.

[21] The Eclipse Project. *The Eclipse Modelling Framework (EMF)*. http://www.eclipse.org/emf/.

[22] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (bpel). *Journal of Grid Computing*, 3(3):283 – 304, September 2005.

[23] A. S. Evans and S. Kent. Meta-modelling semantics of UML: the pUML approach. In *2nd International Conference on the Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science (LNCS)*, pages 140–155. Springer-Verlag, 1999.

[24] D. C. Fallside and P. Walmsley. *XML Schema Part 0: Primer Second Edition*. The World Wide Web Consortium (W3C), October 2004. http://www.w3.org/TR/xmlschema-0/.

[25] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proceedings of the 9th International Workshop on Software Specification and Design*, pages 50–59, 1998.

[26] N. Fenton, W. Marsh, M. Neil, P. Cates, S. Forey, and M. Tailor. Making resource decisions for software projects. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 397 – 406. IEEE Computer Society, May 2004.

[27] W. Frakes and C. Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, 1996.

[28] S. Frolund and J. Koistinen. QML: A language for quality of service specification. Technical Report TR-98-10, HP Laboratories, 1998.

[29] G. Governatori and Z. Milosevic. An approach for validating BCL contract specifications. In G. Governatori and Z. Milosevic, editors, *2nd EDOC Workshop on Contract Architectures and Languages (CoALA 2005)*, September 2005.

[30] GridSAM - grid job submission and monitoring web service. `http://gridsam.sourceforge.net/2.0.1/index.html`.

[31] E. Guerra, P. Díaz, and Juan de Lara. Visual specification of metrics for domain specific visual languages. In *Graph Transformation and Visual Modeling Techniques (GT-VMT'06)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[32] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. In *Computer Performance Evaluation: 10th International Conference, Tools '98*, volume 1469 of *LNCS*, page 51. Springer Berlin, 1998.

[33] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[34] International Business Machines (IBM), Inc. *Web Service Level Agreement (WSLA) Language Specification*, January 2003.

[35] International Standards Organisation (ISO). *Standard Generalized Markup Language (SGML)*, 1986.

[36] International Standards Organisation (ISO). *Reference Model of Open Distributed Processing (RM-ODP)*, June 1995.

[37] The Internet Engineering Task Force (IETF). *Hypertext Transfer Protocol – HTTP/1.1*, June 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[38] The Internet Society. *Uniform Resource Identifier (URI): Generic Syntax*, rfc: 3986 edition, 2005. `http://www.ietf.org/rfc/rfc3986.txt`.

[39] J.-P. Jacquet and A. Abran. Metrics validation proposals: A structured analysis. In *8th International Workshop on Software Measurement*, 1998.

[40] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[41] JANET – The UK's education and research network. `http://www.ja.net/`.

[42] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06), Dijon, France*, pages 1188 – 1195. ACM Press, 2006.

[43] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for Java programs. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[44] The Kent Modelling Framework (KMF). `http://www.cs.kent.ac.uk/projects/kmf/documents.html`.

[45] D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736, 1964.

[46] D. E. Knuth. Literate programming. *The Computer Journal*, 2:97 – 111, May 1984.

[47] C. Kobryn. UML 2001: a standardization odyssey. *Communuications of the ACM*, 42(10):29–37, 1999.

[48] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Program.*, 1(3):26–49, 1988.

[49] D. D. Lamanna, J. Skene, and W. Emmerich. SLAng: A language for service level agreements. In *9th IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 100 – 106. IEEE Press, 2003.

[50] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[51] J. J. Lee and R. Ben-Natan. *Integrating Service Level Agreements*. Wiley Publishing, Inc., 2002.

[52] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder. The Dagstuhl middle metamodel: A schema for reverse engineering. In *International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003)*, volume 94 of *Electronic Notes in Theoretical Computer Science*, pages 7 – 18. Elsevier, 2003.

[53] P. F. Linington. Automating support for e-business contracts. In *Proc. of the EDOC 2004 Workshop on Contract Architectures and Languages*. IEEE Computer Society Press, 2004.

[54] P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kilkarni, and S. Neal. A unified behavioural model and a contract for extended enterprise. In *Data and Knowledge Engineering*, volume 51. Elsevier Science Publishers, 2004.

[55] P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A unified behavioural model and a contract language for extended enterprise. *Data and Knowledge Engineering*, 51(1):5 – 29, October 2004.

[56] G. Lodi, F. Panzieri, D. Rossi, and E. Turrini. SLA-driven clustering of QoS-aware application servers. *IEEE Transactions on Software Engineering*, 33(3):186–197, 2007.

[57] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring quality of service in distributed object systems. In *1st International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 43 – 52. IEEE Press, April 1998.

[58] H. Ma, W. Shao, L. Zhang, Z. Ma, and Y. Jiang. Applying OO metrics to assess UML meta-models. In *The Unified Modelling Language 2004*, volume 3273 of *Lecture Notes in Computer Science*, pages 12 – 26. Springer, 2004.

[59] J. McCarthy. Towards a mathematical science of computation. *Information Processing*, 1962.

[60] J. A. McQuillan and J. F. Power. A definition of the Chidamber and Kemerer metrics suite for the Unified Modeling Language. Technical Report NUIM-CS-TR-2006-03, Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland, 2006.

[61] J. A. McQuillan and J. F. Power. Experiences of using the Dagstuhl middle metamodel for defining software metrics. In *4th International Conference on Principles and Practices of Programming in Java (PPPJ 2006)*, pages 194 – 198, 2006.

[62] J. A. McQuillan and J. F. Power. Towards re-usable metric definitions at the meta-level. In *PhD Workshop of the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, 2006.

[63] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for Web Services*. Prentice Hall, Inc., 2001.

[64] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. In *Graph Grammars Workshops / International Conference on Graph Transformation*, volume 72(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[65] Microsoft Corporation. *COM: Component Object Models Technologies*. `http://www.microsoft.com/com/default.mspx`.

[66] Microsoft Corporation. *Microsoft .Net Homepage*. `http://www.microsoft.com/net/default.mspx`.

[67] Modelling, Simulation and Design Lab, McGill University, Montreal. *AToM^3: A tool for multi-formalism and meta-modelling*. `http://atom3.cs.mcgill.ca/`.

[68] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, 1983.

[69] C. Molina-Jimenez, J. Pruyne, and A. van Moorsel. The role of agreements in IT management software. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 36 – 58. Springer, 2005.

[70] C. Molina-Jimenez, S. Shrivastava, and J. Warne. A method for specifying contract mediated interactions. In *Ninth IEEE International EDOC Enterprise Computing Conference*, pages 106 – 115. IEEE Computer Society, September 2005.

[71] C. Molina-Jimenez, S.K. Shrivastava, E. Solaiman, and J.P. Warne. Contract representation for run-time monitoring and enforcement. In J.-Y. Chung and L.-J. Zhang, editors, *IEEE International Conference on E-Commerce (CEC 2003)*, pages 103 – 110. IEEE Computer Society Press, 2003.

[72] G. Morgan, S. Parkin, C. Molina-Jimenez, and J. Skene. Monitoring middleware for service level agreements in heterogenerous environments. In *Challenges of Expanding Internet: E-Commerce, E-Business, and E-Government. 5th IFIP Conference on e-Commerce, e-Business, and e-Government (I3E 2005)*, volume 189 of *IFIP*, pages 79 – 93. Springer, 2005.

[73] N.Chomsky. *Syntactic Structures*. Mouton, 1957.

[74] S. Neal, J. Cole, P.F. Linington, Z. Milosevic, S. Gibson, and S. Kulkarni. Identifying requirements for business contract language: A monitoring perspective. In M. Steen and B. R. Bryant, editors, *Seventh International Enterprise Distributed Object Computng Conference*, pages 50 – 61. IEEE Computer Society, September 2003.

[75] Jm Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, 1999.

[76] Novosoft LLC. *Novosoft Metadata Framework and UML Library*, 2007. `http://nsuml.sourceforge.net/`.

[77] The Object Management Group (OMG). *The CORBA Trading Service*, formal97-04-01 edition, April 1997.

[78] The Object Management Group (OMG). *Model Driven Architecture (MDA)*, ormsc/01-07-01 edition, July 2001.

[79] The Object Management Group (OMG). *Meta-Object Facility Core Specification Version 2.0*, formal/2006-01-01 edition, April 2002.

[80] The Object Management Group (OMG). *The Meta-Object Facility v1.4*, formal/2002-04-03 edition, April 2002.

[81] The Object Management Group (OMG). *UML 2.0 Superstructure Final Adopted specification*, ptc/03-08-02 edition, 2002.

[82] The Object Management Group (OMG). *UML Profile for Enterprise Distributed Object Computing Specification*, February 2002.

[83] The Object Management Group (OMG). *XML Metadata Interchange (XMI), v1.2*, formal/02-01-01 edition, January 2002.

[84] The Object Management Group (OMG). *Gene Expression, Version 1.1*, formal/2003-10-01 edition, October 2003.

[85] The Object Management Group (OMG). *MDA Guide Version 1.0.1*, omg/2003-06-01 edition, June 2003.

[86] The Object Management Group (OMG). *Meta-Object Facility (MOF) 2.0 Core Proposal*, ad/2003-04-07 edition, April 2003.

[87] The Object Management Group (OMG). *The Unified Modeling Language v1.5*, formal/2003-03-01 edition, March 2003.

[88] The Object Management Group (OMG). *UML 2.0 OCL Final Adopted specification*, ptc/03-10-14 edition, October 2003.

[89] The Object Management Group (OMG). *UML Profile for Schedulability, Performance and Real-time Specification, Final Draft*, ptc/03-03-02 edition, March 2003.

[90] The Object Management Group (OMG). *Common Object Request Broker Architecture: Core Specification*, formal/04-03-12 edition, March 2004.

[91] The Object Management Group (OMG). *Human-Usable Textual Notation (HUTN), V1.0*, formal/2004-08-01 edition, August 2004.

[92] The Object Management Group (OMG). *MOF QVT (Queries/Views/Transformations) Final Adopted Specification*, ptc/05-11-01 edition, November 2005.

[93] The Object Management Group (OMG). *XML Metadata Interchange (XMI), v2.0*, formal/2005-05-01 edition, September 2005.

[94] The Object Management Group (OMG). *XML Metadata Interchange (XMI), v2.1*, formal/2005-09-01 edition, September 2005.

[95] The Object Management Group (OMG). *Diagram Interchange, V1.0*, formal/2006-04-04 edition, April 2006.

[96] The Object Management Group (OMG). *Product Lifecycle Management Services, Version 1.0.1*, formal/2006-04-03 edition, April 2006.

[97] M. Oleneva and W. Beckmann. Application hosting requirements. TAPAS Project Deliverable D1, Adesso AG, Dortmund, September 2002.

[98] Open Middleware Infrastructure Institute UK (OMII). `http://www.omii.ac.uk/`.

[99] Open Grid Forum. *Job Submission Description Language (JSDL) Specification, Version 1.0*, 2005. `http://www.ogf.org/documents/GFD.56.pdf`.

[100] Open Grid Forum. *Web Services Agreement Specification (WS-Agreement) Version 2005/09*, 2006. http://www.ogf.org/Public_Comment_Docs/Documents/Oct-2005/WS-AgreementSpecificationDraft050920.pdf.

[101] Organization for the Advancement of Structured Information Standards (OASIS). *Universal Description Discovery and Integration (UDDI)*, July 2002.

[102] Organization for the Advancement of Structured Information Standards (OASIS). *Web Service Business Process Execution Language Version 2.0*, May 2006. http://www.oasis-open.org/committees/download.php/18714/wsbpel-specification-draft-May17.htm.

[103] A. Paschke. RBSLA - a declarative rule-based service level agreement language based on RuleML. In *International Conference on Intelligent Agents, Web Technology and Internet Commerce (IAWTIC 2005)*, 2005.

[104] A. Paschke. ECA-RuleML/ECA-LP: A homogeneous event-condition-action logic programming language. In *International Conference of Rule Markup Languages (RuleML'06)*, 2006.

[105] K. Patel. XML grammar and parser fro the web service offerings language. Master's thesis, Ottawa-Carleton Institute for Electrical and Computer Engineering, January 2003.

[106] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[107] Python Software Foundation (PSF). *The Python Programming Language*, 2007. http://www.python.org/.

[108] F. Raimondi, W. Emmerich, and J. Skene. A methodology for on-line monitoring non-functional specifications of web-services. In *PROVECS. TOOLS Europe 2007.*, page To appear. ETH Zurich, 2007.

[109] M. Richters and M. Gogolla. On formalizing the UML Object Constraint Language (OCL). In *17th International Conference on Conceptual Modelling (ER'98)*, volume 1507 of *Lecture Notes in Computer Science (LNCS)*, pages 449 – 464. Springer-Verlag, 1998.

[110] The Rule Markup Initiative. http://www.ruleml.org/.

[111] A. Sahai, A. Durante, and V. Machiraju. Towards automated SLA management for web services. Technical Report HPL-2001-310R1, HP Laboratories, 2001. http://www.hpl.hp.co.uk/techreports/2001/HPL-2001-310R1.html.

[112] A. Sahai, V. Machiraju, M. Sayal, L. j. Jin, and F. Casati. Automated SLA monitoring for web services. Technical Report HPL-2002-191, HP Laboratories, 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-191.html.

[113] A. ShaikhAli, O. F. Rana, R. Al-Ali, and D. W. Walker. UDDIe: An extended registry for web services. In *Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, page 85. IEEE Press, 2003.

[114] J. Skene. Implementation of tools for monitorability analysis, 2006. `http://uclslang.sourceforge.net/monitorability.html`.

[115] J. Skene and W. Emmerich. Model driven performance analysis of enterprise information systems. In *Workshop on Test and Analysis of Component Based Systems (TACoS '03), in conjunction with ETAPS '03*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier Science B. V., April 2003.

[116] J. Skene and W. Emmerich. Generating a contract checker for an SLA language. In *Proc. of the EDOC 2004 Workshop on Contract Architectures and Languages*. IEEE Computer Society Press, 2004.

[117] J. Skene and W. Emmerich. Engineering runtime requirements-monitoring systems using MDA technologies. In *Trustworthy Global Computing, International Symposium (TGC 2005)*, volume 3705 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2005.

[118] J. Skene and W. Emmerich. Specifications, not meta-models. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 47–54. ACM Press, 2006.

[119] J. Skene, D. D. Lamanna, and W. Emmerich. Precise service level agreements. In *26th International Conference on Software Engineering (ICSE)*, pages 179–188. IEEE Press, May 2004.

[120] J. Skene, A. Skene, J. Crampton, and W. Emmerich. The monitorability of service-level agreements for application-service provision. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 3–14. ACM Press, 2007.

[121] The SLAng open-source project. `http://uclslang.sourceforge.net/`.

[122] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4), 1994.

[123] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages – A laboratory-based approach*. Addison Wesley, 1995.

[124] E. Solaiman, C. Molina-Jimenez, and S. Shrivastava. Model checking correctness properties of electronic contracts. In *International Conference on Service Oriented Computing (ICSOC03)*, volume 2910 of *Lecture Notes in Computer Science*, pages 303 – 318. Springer, 2003.

[125] R. Staehli, F. Eliassen, J. O. Aagedal, and G. Blair. Quality of service semantics for component-based systems. In *2nd International Conference on Reflective and Adaptive Middleware Systems*,

volume 2672 of *Lecture Notes in Computer Science (LNCS)*, pages 153 – 157. Springer-Verlag, June 2003.

[126] R. Sturm, W. Morris, and M. Jander. *Foundations of Service Level Management*. SAMS, 2000.

[127] Sun Microsystems, Inc. *Java 2 Enterprise Edition*. `http://java.sun.com/j2ee/index.jsp`.

[128] Sun Microsystems, Inc. *Java API for XML-Based RPC (JAX-RPC)*. `http://java.sun.com/webservices/jaxrpc/index.jsp`.

[129] Sun Microsystems, Inc. *Java Metadata Interface JMI specification*. `http://java.sun.com/products/jmi/`.

[130] Sun Microsystems, Inc. *Java Server Pages JSP v. 2.0 specification*. `http://java.sun.com/products/jsp/`.

[131] Sun Microsystems, Inc. *Enterprise Java-Beans (EJB) Specification v2.0*, August 2001.

[132] V. Tosic. *Service Offerings for XML Web Services and Their Management Applications*. PhD thesis, Ottawa-Carleton Institute for Electrical and Computer Engineering, 2002.

[133] V. Tosic, B. Esfandiari, B. Pagurek, and K. Patel. On Requirements for Ontologies in Management of Web Services. In *Web Services, e-Business, and the Semantic Web — CAiSE 2002 Int. Workshop, WES 2002, Toronto, Canada*, Lecture Notes in Computer Science (LNCS), pages 237 – 247. Springer-Verlag, June 2002.

[134] V. Tosic, K. Patel, and B. Pagurek. Reusability constructs in the Web Service Offerings Language (WSOL) [revised extended addition]. Technical Report SCE-02-14, Department of Systems and Computer Engineerin, Carleton University, Ottawa, May 2003.

[135] The UCL MDA tools. `http://uclmda.sourceforge.net/`.

[136] É. Vépa, J. Bézivin, H. Brunelière, and F. Jouault. Measuring model repositories. In *Model Size Metrics Workshop at MoDELS/UML 2006*, 2006.

[137] G. H. von Wright. Deontic logic. *Mind*, 60:1 – 15, 1951.

[138] W3C. *Hyper-Text Markup Language 4.01 Specification*, December 1999. `http://www.w3.org/TR/html401/`.

[139] Wikipedia. *Chebychev's Inequality*, 2006. `http://en.wikipedia.org/wiki/Chebyshev%27s_inequality`.

[140] The World Wide Web Consortium (W3C). *Mathematical Markup Language (MathML) Version 2.0 (Second Edition)*, October 2003. `http://www.w3.org/TR/2003/REC-MathML2-20031021/`.

[141] The World Wide Web Consortium (W3C). *Scalable Vector Graphics (SVG) 1.1 Specification*, January 2003. http://www.w3.org/TR/SVG/.

[142] The World Wide Web Consortium (W3C). *EXtensible Markup Language (XML) 1.0 (Third Edition)*, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

[143] The World Wide Web Consortium (W3C). *OWL Web Ontology Language Overview*, February 2004. http://www.w3.org/TR/owl-features/.

[144] The World Wide Web Consortium (W3C). *RDF Primer*, February 2004. http://www.w3.org/TR/rdf-primer/.

[145] The World Wide Web Consortium (W3C). *Web Services Architecture*, February 2004. http://www.w3.org/TR/ws-arch/.

[146] The World Wide Web Consortium (W3C). *OWL-S: Semantic Markup for Web Services*, 2004 November. http://www.w3.org/Submission/OWL-S/.

[147] R. Yin. *Case Study Research: Design and Methods*. SAGE Publications, 3rd edition, 2003.

[148] A. Avižienis, J.-C. Laprie, and B. Randall. Fundamental concepts of dependability. In *3rd IEEE Information Survivability Workshop (ISW-2000)*, pages 7 – 12, October 2000.