Engineering Runtime Requirements-Monitoring Systems using MDA Technologies*

James Skene and Wolfgang Emmerich

Dept. of Computer Science, University College London
Gower St, London, WC1E 6BT, UK
{j.skene|w.emmerich}@cs.ucl.ac.uk

Abstract. The Model-Driven Architecture (MDA) technology toolset includes a language for describing the structure of meta-data, the MOF, and a language for describing consistency properties that data must exhibit, the OCL. Off-the-shelf tools can generate meta-data repositories and perform consistency checking over the data they contain. In this paper we describe how these tools can be used to implement runtime requirements monitoring of systems by modelling the required behaviour of the system, implementing a meta-data repository to collect system data, and consistency checking the repository to discover violations. We evaluate the approach by implementing a contract checker for the SLAng service-level agreement language, a language defined using a MOF meta-model, and integrating the checker into an Enterprise JavaBeans application. We discuss scalability issues resulting from immaturities in the applied technologies, leading to recommendations for their future development.

1 Introduction

Run-time monitoring of systems is useful in a variety of situations in which the behaviour of a system cannot be guaranteed in advance. Such situations include testing a system against its requirements if it cannot be proven to meet them by construction, or monitoring the behaviour of a system where the actions of external agents, such as its users, is the actual object of scrutiny. Such monitoring can be used in conjunction with a contractual agreement to establish a strong basis for trust in a system: the owners of the system agree that it will behave in a particular way, and the system is monitored to ensure that deviations from the desired behaviour are detected and properly compensated for.

In the past, several approaches to the automatic implementation of runtime requirements-monitoring systems have been proposed. Such automatic implementation is intended to provide control over the specification of monitoring, improve the accuracy of monitoring and reduce the cost of its implementation. This paper presents a novel approach to runtime requirements monitoring that has arisen out of work to develop a contract checker for a service-level agreement

* This work was partially funded by the TAPAS project, IST-2001-34069.

(SLA) language, SLAng. The approach relies on several standards published by the Object Management Group (OMG), and appears to be particularly suitable for comparing the behaviour of a system to sets of requirements that can be selected dynamically at runtime from a range of possible options, as is typical in SLAs.

The Model-Driven Architecture (MDA) technology toolset includes a language for describing the structure of meta-data, the Meta-Object Facility (MOF), and a language for describing consistency properties that data must exhibit, the Object Constraint Language (OCL). The Java Meta-Data (JMI) standard prescribes patterns for implementing programmatic access to MOF defined meta-data repositories. These patterns can be implemented in a generative programming tool to generate implementations of repositories, which can in turn be integrated with off-the-shelf tools to perform consistency checking over the data they contain.

The SLAng SLA language is defined using a MOF meta-model that models the required behaviour of electronic services governed by SLAs. The model is divided into two parts, the first describing the syntactic structure of SLAng contracts, the second describing the behaviour of the services that the contracts govern. Associations and OCL constraints between the two parts serve to specify the semantics of the language, both by associating SLAs with the services to which they apply, and by describing the restrictions on the behaviour of those services that the SLAs imply. The original intent of this approach was to provide a precise definition of the language. However, in combination with the JMI mapping and an OCL interpreter, the meta-model serves as a specification from which a contract checker can be generated. This contract checker can be combined with simple hand-implemented software instruments to form a complete runtime monitoring system.

In this paper we describe the approach, critically discuss it as an alternative to previous work on runtime monitoring, and report on our practical experience with the technologies involved. The paper includes an overview of the approach in Section 2. In Section 3 we briefly review the features of the SLAng language and its specification. In Section 4 we discuss the design and implementation of a tool for generating the checker. In Section 5 we describe the architecture of the resulting checker. In Section 6 we describe the deployment of the checker to monitor an Enterprise JavaBeans application, and evaluate the practicality of the approach. In Section 7 we compare the approach to other work on run-time monitoring. Finally, in Section 8 we make some concluding remarks, and discuss future work.

2 Runtime requirements monitoring using MOF and OCL

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 requires data gathered over an extended period. In the approach outlined in this paper the requirements checking logic and repository are implemented using a combination of automatic code generation from a MOF model and a reusable OCL checker component. Generating software instruments is discussed below.

MOF models are very similar to UML class models [22]. They include sets of classes, the data they contain, and their relationships. Constraints on the model that cannot be represented graphically are expressed using the OCL. OCL is a typed-expression language similar to the expressions parts of Java or C++, and is used to describe class invariants in the model.

The classes in a MOF model can be interpreted as directly modelling objects in the real world, as is the case in the SLAng meta-model which describes the way that services should behave in the presence of SLAs. Requirements can be expressed directly as constraints over the behaviour of services, which will generally be modelled as classes of events arising during the execution of the service. Alternatively they can be expressed in the context of a model of a requirements language associated with the service. Instances of this model are requirements that may be expressed in the language at runtime and associated with services. Constraints between the model of the requirements language and the model of the service describe how the service must act in the presence of the requirements. In this manner the semantics of the SLAng SLA language are defined in terms of constraints over the performance of services that only apply when SLAs are present.

The meta-model can alternatively be interpreted as a model of data describing the world, and the set of conditions necessary for those data to meet some set of requirements. If we interpret the meta-model in this way, then we can produce a computer program capable of holding those data and checking them, to see whether services are behaving in the way that we want them to.

This approach is shown in Figure 1 in which thick arrows represent code generation, and thin arrows represent data flow. The figure represents the case in which a requirements language is being used to specify requirements at runtime.

To implement the approach we found it necessary to develop a JMI generator. (As discussed in Section 4, this was needed because previous generators did not offer adequate flexibility over the type of code generated. However, this component may be considered 'off-the-shelf' as it is a standard MDA component independent of the particular application.) We combined the resulting generated data structures with the OCL2 interpreter implemented at Kent University [12], which features an extension allowing it to evaluate OCL constraints over plain Java objects using Java reflection. The design of the JMI generator is discussed in more detail in the next section. The design of the resulting checker is discussed in detail in Section 5. The performance of the checker is described in 6.

A complete requirements monitoring system also includes software instruments to gather the event data included in the service model. The implementation of these instruments requires the interpretation of the service model in the context of the particular system being instrumented. If the service model is de-

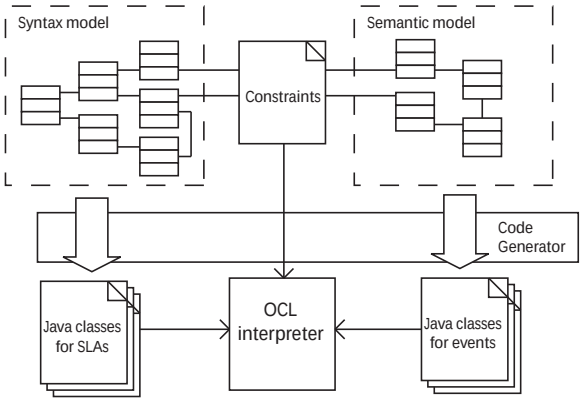

Fig. 1. Generating an SLA checker from the SLAng meta-model

scribed in the same terms as the system being monitored, for example in terms of particular Java classes and operations present in the implementation, then it will be possible to implement a generator for the instruments directly for the model. However, it may be that the service model is at a higher level of abstraction, and intended to apply to services with a range of designs and implementation technologies, as is the case with the SLAng language used as an example in this paper. In this case the instruments must be implemented manually, although the explicit nature of the service model provides considerable guidance in this process. In summary, the possibility of generating the instruments automatically depends on the level of abstraction of the MOF model, although we have not yet investigated the generation of instruments in practice.

3 The SLAng language

The SLAng language syntax and semantics are defined by a MOF (version 1.1) model [20]. The model provides a formal definition of the structure of the syntax of the language, and of the semantic domain in which SLAs apply. These are modelled in terms of classes of objects with attributes and associations. Constraints in the model restrict the sets of objects described so that SLAs are only ever associated with services that are consistent with their terms and which meet their conditions. In this way the semantics of the language are formally defined. This approach was inspired by the work of the Precise UML group (pUML), who used the approach to define the semantics for their UML 2 submissions [13].

A view of the meta-model showing the syntax of the Electronic Service (ES) SLA is shown in Figure 2. The SLA is divided into a section for defining terms, and another for conditions. The conditions section is further subdivided between conditions on the behaviour of the service provider, and conditions on the behaviour of the client.

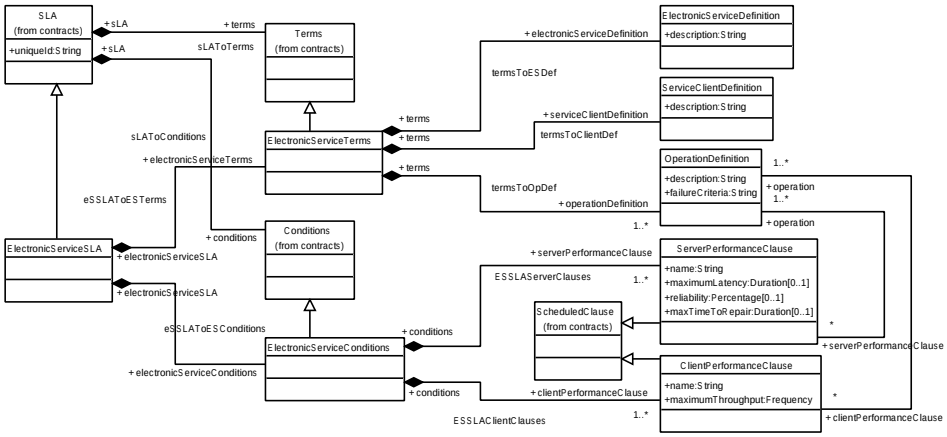

Fig. 2. Model of the syntax of SLAng electronic-service contracts

The use of a MOF meta-model to define the syntax of SLAng confers the advantages of the XML Metadata Interchange (XMI) [21] standard, a standard for serialising MOF-defined metadata. The XMI mapping of the SLAng syntactic model constitutes the concrete syntax of the language.

The semantic model of electronic service provision is shown in Figure 3. Service usages are events, occurring over a period, with the possibility of failure. They are associated with an operation, which forms part of an electronic service. They are also associated with the client that caused the usage. The syntactic and semantic models are co-located in a single model, and the terms in the syntactic model are associated with elements in the semantic model in order to define their meaning.

As stated above, the SLAng meta-model also includes OCL constraints that give meaning to condition statements in the language. The following is the top-level invariant defining the meaning of performance and reliability for Electronic Service SLAs:

context contracts::es::ServerPerformanceClause **inv**:
operation→collect(o : contracts::asp::OperationDefinition |
o.operation
)→forAll(o : services::Operation |
observedDowntime(o) < (timeRemaining(-1) ⋆ (1 - reliability)))

This expression is explained in detail in [23][1]. It relies on a number of function definitions, such as `observedDowntime` defined in the specification. The total amount of OCL for this constraint runs to about 50 lines.

[1] The expression is slightly modified from [23] as a result of testing and developing the meta-model and constraints using the generated SLA checker. However, its intent is the same and its structure is quite similar.

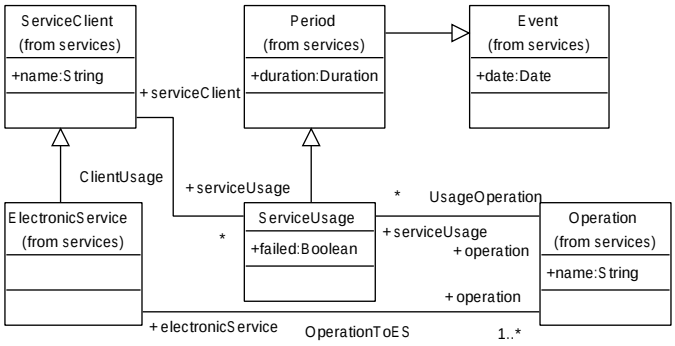

Fig. 3. Model of electronic service usage

In this section we have presented an overview of the SLAng language and its specification. For a more detailed discussion of the language, including a discussion of design decisions and objectives, and a comparison to other SLA languages and technologies, please refer to [23].

4 A JMI Generator

The JMI generator is implemented in Java, and follows the design shown in Figure 4. It is heavily dependent on the Velocity Template Engine (VTE) [11], developed as part of the Apache project. Similar to Java Server Pages (JSP) [5], or PHP [7], Velocity is a tool for generating text from predefined templates. These templates are text files that include fields delimited using special characters. The VTE is configured with these templates, and also extra data called 'context'. The templates are parsed by the VTE: ordinary text is passed straight through; the fields in the templates either control the order of parsing, for example by specifying optional or repeated sections, or indicate that data from the context should be inserted. By varying the context, several outputs can be produced from the same template.

The templates in our implementation are taken from the JMI specification, and translated into Velocity's template syntax. The JMI specification requires Java types to be produced corresponding to elements in the metamodel: for each class, a 'class proxy' interface, for creating and finding instances of the class, and an 'instance' interface, for editing properties and invoking operations of instances of the class, are required; for associations, an 'association proxy' interface for creating and querying pairs of associated instances; for each package, 'package proxy' interface enabling the discovery of class proxies, association proxies and subpackage proxies; for enumerations, an interface type for enumeration values and a class containing static exemplars of enumeration values. The JMI standard also specifies XMI reader and writer interfaces.

The generator includes a template for each of these types. Except in the case of enumerations, the JMI specification only defines interfaces, but does not

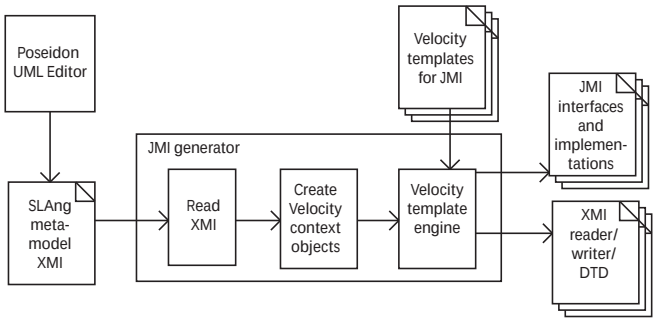

Fig. 4. Design of the JMI generator

indicate how they are to be implemented. The generator therefore also includes templates for implementations of each of the above elements. The generator also has a template to produce an XMI DTD following the pattern described in the XMI standard.

The context for each of these templates is drawn from the particular MOF model for which a set of JMI interfaces is being generated. In our case this is the SLAng meta-model. The meta-model is exported from a modelling tool in an XMI format file. The first stage of the JMI generator reads this file and creates an in-memory representation of it.

This initial in-memory representation of the API is not a suitable context for the Velocity templates, as it reflects the structure of the XMI file, rather than the structure of the templates. Velocity templates can only perform quite simple data manipulation (they lack recursion, for example, which makes it difficult to navigate data structures in the context). They must therefore be supplied with their context data in a form that closely reflects the way it is used in the template. The second stage of the generator creates a number of different context objects, appropriate to the Java files that must be generated, using the data from the in-memory representation of the XMI file.

In the third stage of its operation, the VTE is invoked using the generated context objects and the JMI templates, in order to generate the requisite JMI Java code. This is placed in the appropriate places in a package directory hierarchy on the file system.

Generating program code from UML diagrams is an important step in the Model Driven Architecture (MDA) methodology. A number of systems to achieve this have been developed with varying degrees of flexibility in the specification of their output. However, we found none to be ideal for our purposes, and elected to implement a generator by hand instead. We evaluated a number of tools in the autumn of 2003 before deciding on this course. These included the Netbeans Meta-Data Repository (MDR) [10], the Eclipse Modelling Framework (EMF) [8], and Novosoft's NSUML [6]. The EMF was rejected because it generates non-standard code from a non-standard meta-model (i.e. not JMI from MOF). The MDR and Novosoft were rejected because at the time they manifested prob-

lems reading standard XMI as generated by our modelling tool of choice. We also wished to reserve the possibility of modifying the JMI implementation code generated by our system, and both of these systems require code-level modifications to alter the generated JMI implementations, reducing the benefits of reuse considerably.

The architecture of the AndroMDA tool [1] is essentially identical to that presented here. However, as stated above, Velocity templates do not have powerful control structures and without the ability to modify the structure of the context objects to preprocess model information it is impossible to generate some outputs. We found the OCL-based approach of the Kent Modelling Framework, version 3 [9] to be adequately expressive. However, the OCL expressions are hard to write when a 'generation state' has to be maintained, containing things like a list of unique identifiers used. For this reason we preferred to use more conventional templates.

The decision not to reuse an existing modelling framework was an engineering descision. In principle any of the systems mentioned above could be adapted to our approach with some degree of effort. However, our requirement of flexibility in the generation of the implementation of the system will probably turn out to be a general requirement, because, as discussed in the evaluation section of this paper, modelling frameworks of this kind will need be adaptable to meet application-specific scalability requirements.

5 Architecture of the SLA checker

The SLA checker consists 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.

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.

To demonstrate the SLA checker and to assist in the development of the SLAng semantics, we have implemented a browser that allows the editing of SLA and event data, via a tree-view of the model.

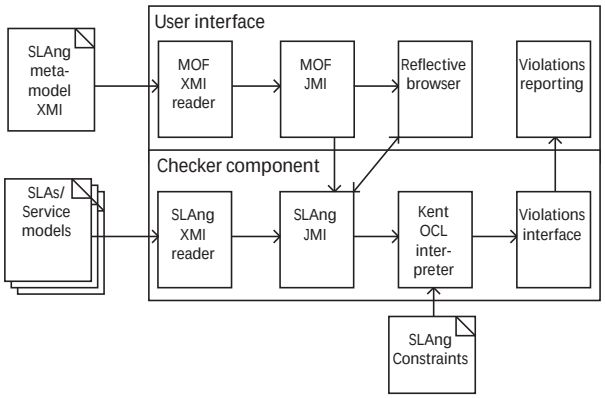

Fig. 5. Design of the SLA checker

The user-interface also allows interactive editing and checking of the constraints over the SLAng model, possible because the OCL constraints are interpreted at runtime, rather than compiled into the implementation language, Java.. The design of the checker is shown in Figure 5. A screenshot of the user interface is shown in Figure 6. The leftmost panel in the user interface contains the tree representing the SLAng model (SLAs and events). The middle panel lists the constraints over the model, and the rightmost panel allows the editing of constraints.

6 Evaluation

6.1 Deployment of the SLA checker

We tested the SLA checker by deploying it to monitor the performance of an EJB application. The application is an auction management system developed

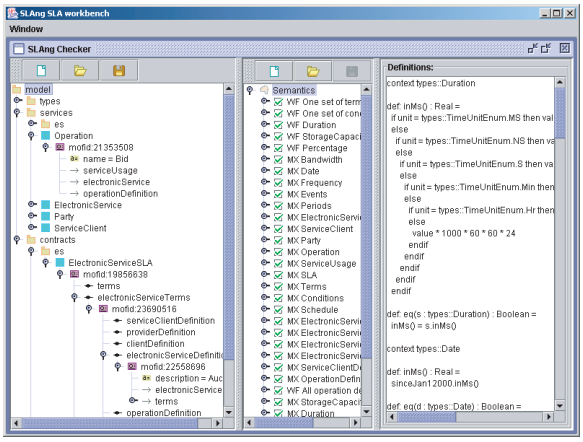

Fig. 6. Screenshot of the SLA checker user interface

by an industrial collaborator. The application is deployed in the popular application server JBoss, which implements the Java 2 Enterprise Edition (J2EE) specification [4], using Apache Tomcat to serve the web front-end [2].

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. We have deployed the SLA checker 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, we 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.

Service performance information is passed to the SLAng service by a server-side interceptor configured as an option of the JBoss container configuration. JBoss remoting operates 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, we added an interceptor on the server side to measure time spent processing EJB requests. The interceptor accesses the SLAng service using JNDI

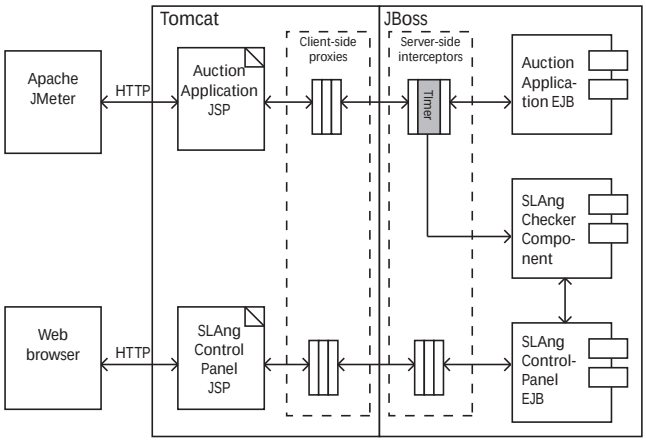

Fig. 7. The SLA checker component deployed to monitor an EJB application

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 [3].

6.2 Results

In this section we evaluate the SLA checker 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 could be speciously discounted on the grounds that they are separate efforts from the implementation of the actual component. If this were the case, then implementing the component would have taken around 1 man-week of labour. In fact, the total amount of labour has been closer to 1 man-year, and JMI generator, language and component have co-evolved to some extent. Indeed, as discussed below, the JMI generator, or at least it's templates will have to continue to adapt in the face of performance requirements that are somewhat related to the domain of the application, i.e. checking SLAng contracts. 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. This is manifest in two ways: Firstly, and most seriously, the time taken to evaluate the OCL constraints is highly correlated to the size of the model, and is far too long for models containing realistic amounts of service data. For a data set of 1000 service usages, the client throughput constraint compares every pair of usages to determine if they occur too closely together. If none do, this results in a million comparisons, and takes 20 minutes on a PC with 1.7GHz Intel Pentium 4 processor. The evaluation is slow due to a combination of factors: The OCL interpreter performs almost no optimisations, the interpretation of the OCL is innately expensive, and the data model over which the expressions are evaluated offers no shortcuts, such as indices.

The second issue is related. In our current implementation of the JMI interfaces all data is represented as Java objects stored in main memory. Since we have implemented no policy for removing or persisting old data, 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.

To correct these issues without discarding the approach altogether requires some reengineering. The data model needs to be backed by a database. This could be either object oriented, or the translation to a more conventional model could be managed by the generated Java code for a particular model. Clearly not all data can be assumed to be in memory at the same time, and this may need to be reflected in the interface to the model data. The evaluation speed of the OCL constraints could be improved by translating it to Java, or possibly SQL (with some reduction in expressive power), rather than interpreting it. We gained some improvement in evaluation time by adding results caching to the OCL interpreter. Further optimisation of evaluation is required, and if the constraints are still to be evaluated across a generated interface, the generated interface may have to provide indices to assist in evaluation, possibly resulting in a closer coupling between interface standard and OCL evaluator.

Clearly these refinements should be the subject of further research.

7 Related work

In [23] we provide a detailed comparison of SLAng with previous SLA languages, focusing on the extent to which these languages provide explicit definitions of

their terms and conditions. Our use of an explicit model for this seems to be quite novel, and it is this feature of the language that allows us to generate the checker automatically.

A similar approach has been proposed in [19], 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 Buisiness Contract Language (BCL) [18], 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 metamodels 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.

Various other systems effectively define their own meta-models for requirements. Representative examples are: the Java-MaC system [16] which automatically embeds monitors in Java code from a requirements specification written in a language called PEDL/MEDL; Java PathExplorer [15] which does the same, but allows requirements to be specified in any high-level logic compatible with the Maude rewriting engine; and the KAOS-FLEA [14] 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 MOF/OCL to describe constraints on a system. JavaMaC and Java PathExplorer are examples of systems capable of generating software instruments thanks to the fact that their semantics are at least partially defined in terms of the structure of Java programs.

MOF/OCL offers the possibility to defer the specification of some requirements until runtime, by specifying requirements in terms of consistency relationships between the system and a model of a requirements language. In this way, the approach can be used to engineer a range of monitoring solutions, each with a language appropriate to their particular needs. This is in contrast to the approaches mentioned above, which prescribe a language for requirements, with the exception of Java PathExplorer which prescribes that a logic be used.

Choosing between systems for runtime requirements management requires at least two questions to be answered: in what form do I wish to represent my requirements? and, which monitoring technology will be practical? In comparison to other approaches, the use of MOF/OCL is very general, but also quite well aligned with conventional software engineering practice in that it is very similar to the use of UML. It is practical in the sense that it can be imple-

mented using off-the-shelf technologies, but impractical in the sense that those technologies currently do not scale well (they may do in the future). In contrast, the approaches listed above assume the existence of a bespoke module implementing the logic for checking for violations. Engineering this module seperately may assist in scalability, although a more efficient OCL interpreter could equally easily be assumed. In terms of investigating the run-time performance of performance monitors, useful work has been done in [17], which demonstrates that the evaluation of requirements can be intractable, depending on the type of the requirement. A more comprehensive survey of the performance and practicality of available technologies would be desirable future work.

8 Conclusion

This paper has described the use of MDA technologies (although not necessarily an MDA approach) to produce runtime requirements monitoring systems. This has been exemplified by our implementation of an SLA checker, automatically, from the specification of our SLA language, SLAng.

In situations in which systems must be monitored against requirements specified at runtime designers may wish to consider adopting the approach as it offers the possibility to generate all or part of an interpreter for a requirements specification language (such as an SLA language) automatically. Where an explicit representation of the semantic primitives of such a language is practical, an OCL interpreter can be employed to check that these semantic elements are consistent with statements in the language, thereby implementing the logical part of a runtime requirements monitoring system. The approach is equally applicable in cases in which the requirements are invariant at runtime – the constraints in the model of the service are simply specified independently of any language model.

Our evaluation of the checker revealed some serious practical issues arising from immaturities in the technologies employed. Although for restricted numbers of objects the implementation serves its purpose, it seems that to achieve scalability both the mapping to implementation and the implementation of off-the-shelf components such as the OCL interpreter must be considerably more sophisticated. This is a consideration beyond SLA checking, as it is reasonable to assume that large software development efforts will wish to maintain and check consistency within large repositories of models. Future research should investigate this mapping further to produce implementation prescriptions to complement interface standards such as the JMI.[2]

References

1. AndroMDA code generation tool. `http://www.andromda.org/`.

[2] Thanks to Werner Beckmann and Addesso, Inc. for the auction application. Also thanks to our TAPAS partners for their input into this work, and to Marc Fleury for his advice concerning JBoss.

2. Apache Jakarta Tomcat servlet container. http://jakarta.apache.org/tomcat/.
3. Apache JMeter. http://jakarta.apache.org/jmeter/.
4. Java 2 Enterprise Edition. http://java.sun.com/j2ee/index.jsp.
5. Java Server Pages JSP v. 2.0 specification. http://java.sun.com/products/jsp/.
6. Novosoft Metadata Framework and UML Library (NSUML). http://nsuml.sourceforge.net/.
7. PHP: PHP Hypertext Preprocessor. http://www.php.net/.
8. The Eclipse Modelling Framework (EMF). http://www.eclipse.org/emf/.
9. The Kent Modelling Framework (KMF). http://www.cs.kent.ac.uk/projects/kmf/documents.html.
10. The Netbeans Meta-Data Repository (MDR) Project. http://mdr.netbeans.org/.
11. The Velocity Template Engine v1.4. http://jakarta.apache.org/velocity/.
12. David Akehurst, Peter Linington, and Octavian Patrascoiu. OCL 2.0: Implementing the Standard. Technical report, Computer Laboratory, University of Kent, November 2003.
13. 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, Colorado, USA, 1999. Springer-Verlag.
14. 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.
15. Klaus Havelund and Grigore Rosu. Monitoring java programs with java pathexplorer. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
16. Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-mac: a run-time assurance tool for java programs. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
17. Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring - fundamentals of java-mac. In *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier Science Publishers, 2002.
18. 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.
19. Peter F. Linington. Automating support for e-business contracts. In *Proc. of the EDOC 2004 Workshop on Contract Architectures and Languages, Monterey, California*. IEEE Computer Society Press, 2004.
20. The Object Management Group (OMG). *The Meta-Object Facility v1.4*, formal/2002-04-03 edition, April 2002.
21. The Object Management Group (OMG). *XML Metadata Interchange (XMI), v1.2*, formal/02-01-01 edition, January 2002.
22. The Object Management Group (OMG). *The Unified Modelling Language v1.5*, formal/2003-03-01 edition, March 2003.
23. J. Skene, D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proc. of the 26th Int. Conference on Software Engineering, Edinburgh, UK*, pages 179–188. IEEE Computer Society Press, May 2004.