Generating a Contract Checker for an SLA Language

James Skene, Wolfgang Emmerich
Dept. of Computer Science, UCL
Gower St, London WC1E 6BT
{j.skene|w.emmerich}@cs.ucl.ac.uk

***Abstract*— SLAng is a language for expressing Service Level Agreements (SLAs) under development as part of the European project TAPAS. It is defined using a meta-model, an instance of the Meta-Object Facility (MOF) model, in which the relationship between the syntax of the language and its domain of application is explicitly represented, and the violation semantics of the language defined using Object Constraint Language (OCL) constraints. The concrete syntax of the language is the XML Meta-data Interchange (XMI) mapping of the syntactic part of the meta-model. In this paper we describe how the Java Meta-data Interface (JMI) mapping can be applied to the meta-model of the language to generate interfaces and classes to create and query SLAs and relevant service monitoring data in memory; and how an OCL interpreter can be applied to check violation constraints over this data, resulting in the implementation of a contract checker that is highly likely to respect the semantics of the language.**

***Index Terms*— Service level agreements, contracts, generative programming, UML, MOF, MDA.**

I. Introduction

IN [1] we introduced SLAng, a language for Service Level Agreements (SLAs). An SLA is the part of a contract between the client and provider of a service that defines the parties' obligations with respect to the qualities of the service, usually taken to mean its performance and reliability. This first paper documented two novel features of the language: it is scoped according to an informal reference model of distributed systems' architecture, so it predefines syntax for the types of agreements likely to be useful in the context of today's Internet; and it explicitly includes client responsibilities, in recognition of the fact that the provider must be protected from malicious behaviour on the part of the client as much as the client must be protected from failures on the part of the provider to deliver requisite levels of service.

The principle requirement of an SLA is to define unambiguously the obligations of the parties in a particular service provision scenario. When a party fails to meet these obligations, a violation is said to have occurred. Clearly, if disagreements over violations are possible, then the utility of an SLA is significantly diminished. Financial penalties are often associated with violations, in order to mitigate the risk to the injured party that such violations imply. Fraud, either accidental or malicious, is possible if violations cannot be proven to have occurred with a high degree of confidence.

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

An SLA written in a pre-defined language such as SLAng relies on the definition of the language for part of its meaning. In [2] we described modifications to the definition of SLAng to improve its precision with respect to the definition of violations. The parties to an SLA can disagree over whether a violation has occurred in at least four different ways:

1) They can disagree over the terms of the agreement, by disagreeing over whether a particular piece of monitoring data or aspect of the services configuration is relevant to the calculation of a violation.
2) They can disagree over the conditions of the agreement, by disagreeing over whether a particular behaviour of the service constitutes a violation.
3) They can disagree over the amount of error introduced by the particular process or mechanism for calculating whether a violation has occurred from a particular set of monitoring data.
4) They can disagree about the amount of error present in any monitoring data, in effect a disagreement over the degree to which a particular set of monitoring data represents the true behaviour of the service.

The contribution of our second paper was to address the first two types of disagreement listed above. We achieved this by applying a meta-modelling technique to the definition of the language, in which both the syntax and semantic domain of the language are explicitly modelled using a Meta-Object Facility (MOF) model (similar to a UML class diagram). The syntactic part of the model defines the format of SLAng SLAs. The semantic part of the model can be interpreted as describing the objects and events in the real world to which the syntactic elements refer, in this case service infrastructure and the events associated with service provision.

The co-location and association of syntactic and semantic elements in the language meta-model significantly reduces the ability of the parties to disagree over the meaning of terms in the language, as the syntactic elements are associated with semantic elements that disambiguate their meaning.

To ensure that the conditions of the SLA are also unambiguous, the model contains constraints over the associations between syntactic and semantic elements. These ensure that SLA statements are only associated with behaviour (represented by the semantic model) that is acceptable according to the quantities specified in the SLA. The constraints hence define the meaning of conditions for SLAng SLAs. They are

expressed in OCL [3], a language with formal semantics of its own, and so are unambiguous, thereby addressing the second cause of disagreements above. The meta-model can be thought of as a model of a world in which all SLAs are respected by the parties to them.

This paper describes the way in which the language meta-model and associated constraints can be used as the input for a generative programming tool to automatically generate a contract checker. The checker compares the measured performance of a service with a set of SLAs to determine if violations have occurred. By automatically generating the checker from the specification of the language semantics, no human errors of interpretation can be introduced in the process of implementing the checker. This makes it harder to dispute the output of the checker on the grounds that it does not respect the language specification and therefore the intent of the SLAs. This reduces the possibility of the third type of disagreement from the list above.

The technologies and process applied to generating the contract checker are components of the Model Driven Architecture (MDA) [4] approach under development by the OMG. This work can be seen as a case study in the advantages of defining domain-specific languages using meta-models, and of generating code automatically from models.

In outline, our paper reads as follows: In Section II we review the features of the SLAng language specification. In Section III we describe in more detail the motivation for generating a checker component automatically, and the approach taken to achieve this. In Section IV we discuss the design and implementation of a tool for generating the checker. In Section V we describe the architecture of the resulting checker. In Section VI we discuss related work. Finally, in Section VII we make some concluding remarks, and discuss future work.

II. Overview of the SLAng language specification

SLAng is defined by a combination of a document, 'The SLAng Specification' [5], and a MOF (version 1.1) model [6]. 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) [7].

The specification document presents views of the model, and describes its elements in English, thereby clearly establishing the meaning of the elements, whose interpretation is otherwise only implied in their natural-language names, structure and relationships. The elements include syntactic elements, which should be interpreted as parts of SLAs, and semantic elements, which should be interpreted as parts or behaviours of services in the real world. The specification document mimics the documentation standards applied by the Object Management Group (OMG) [8] when documenting their meta-model-based standards, such as UML and the MOF.

MOF models are commonly called 'meta-models', because they are used to describe meta-data (data about data, or 'models'). Our original intent of using the MOF was to allow the description of data concerning electronic services, so using the MOF to describe SLAs for electronic services is quite appropriate. We will henceforth refer to the SLAng MOF model as the SLAng 'meta-model'. The term 'SLAng specification' refers to the specification document that describes the model.

MOF models are very similar to UML class models, and in fact we use the UML tool Poseidon [9] to maintain the SLAng meta-model. Poseidon produces UML version 1.4 class models, not MOF models. However, differences between the two standards can be eliminated by using the UML profile for MOF models [10] introduced as part of the EDOC standard profile. This adapts the syntax of the UML slightly, so that it can represent all elements in the MOF meta-model.

When SLAng was initially presented in [1] it could express SLAs for the following kinds of service:

- Application – In which a thin client uses a web- or application-service.
- Hosting – In which components use an execution environment (a container).
- Persistence – In which a container uses a Storage Service Provider (SSP).
- Communication – In which a container uses an Internet Service Provider (ISP).
- Service – In which one application service uses another.
- Container – In which a container uses another for purposes of replication or load balancing.
- Networking – In which ISPs agree to convey traffic across network boundaries.

However, since adopting the meta-modelling approach described in [2], we have only completed the meta-model for Electronic Service SLAs, which are the amalgam of the Application and Service type SLAs previously defined (and found to be so similar, that no distinction was useful between the two). The models and discussions in this paper therefore pertain to ES SLAs only. In future we intend to expand the language to describe SLAs for the other types of services listed above.

A view of the meta-model showing the syntax of the ES SLA is shown in Figure 1. 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.

The semantic model of electronic service provision is shown in Figure 2. It is currently quite simple. Service usages are events, occurring at some instant and having a duration, 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.

Although the model of service usage for application services presented here is simple, it is explicit and fairly unambiguous. It serves as a reference for the definition of terms seen in the syntax of the ASP SLA.

The syntactic and semantic models are co-located in a single model, and the terms in the syntactic model are associated

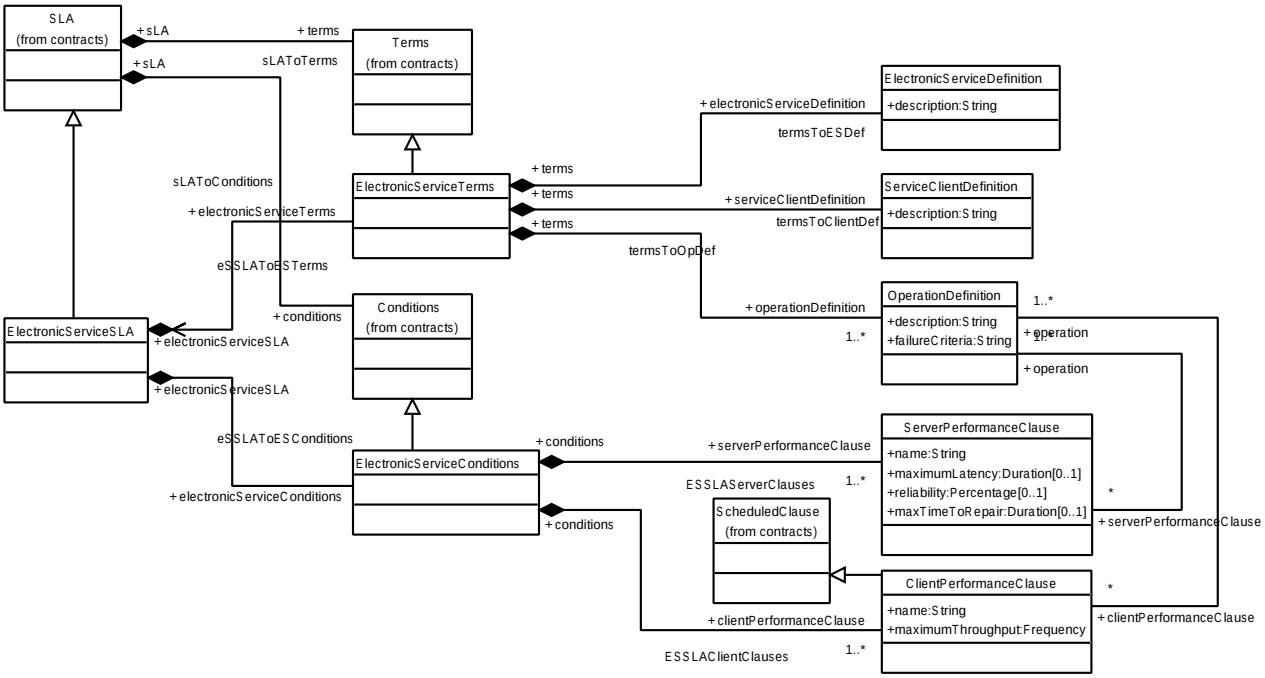

Fig. 1. Model of the syntax of SLAng electronic-service contracts

with elements in the semantic model in order to define their meaning. This relationship is shown in Figure 3.

As stated above, the SLAng meta-model also includes OCL constraints that give meaning to condition statements in the language. These OCL constraints are part of the meta-model, and when we refer to the meta-model subsequently, we will also be referring to the constraints. However, for convenience we maintain them in text files outside of the Poseidon tool used to edit the UML diagrams. The following is the top-level invariant defining the meaning of performance and reliability for ASP 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 [2][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, and may be found in the language specification. The specification also includes a number of constraints that enforce well-formedness of SLAs or eliminate illogical situations from the semantic model.

The concrete syntax of SLAng, used to represent and exchange SLAs, is the XMI mapping of the syntactic part of the meta-model. XMI (the XML Metadata Interchange format) [11] is a standard that specifies a text format for the exchange of models whose structure is defined using a MOF meta-model, by mapping meta-models onto XML Document Type Definitions (DTDs) [12].

[1]The expression is slightly modified from [2] as a result of testing and developing the meta-model and constraints using the generated contract checker. However, its intent is the same and overall structure quite similar.

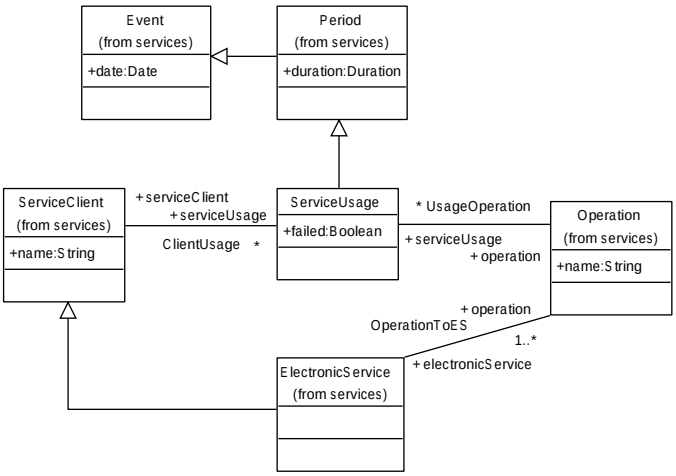

Fig. 2. Model of electronic service usage

III. Generating a contract checker

The SLAng meta-model and constraints, as used in the language specification, are a model of the world as we hope it will be. The model states that in the world a collection of things exist that are called SLAng SLAs, which are structured in a particular way. It further describes a set of things called Electronic services, and the way in which those services can behave. It states that there may be an association between SLAs and the services they govern, and that if this is the case then the behaviour of those services and their clients is restricted so as to be acceptable according to the values specified in the SLA.

The first idea in this paper is that the meta-model can

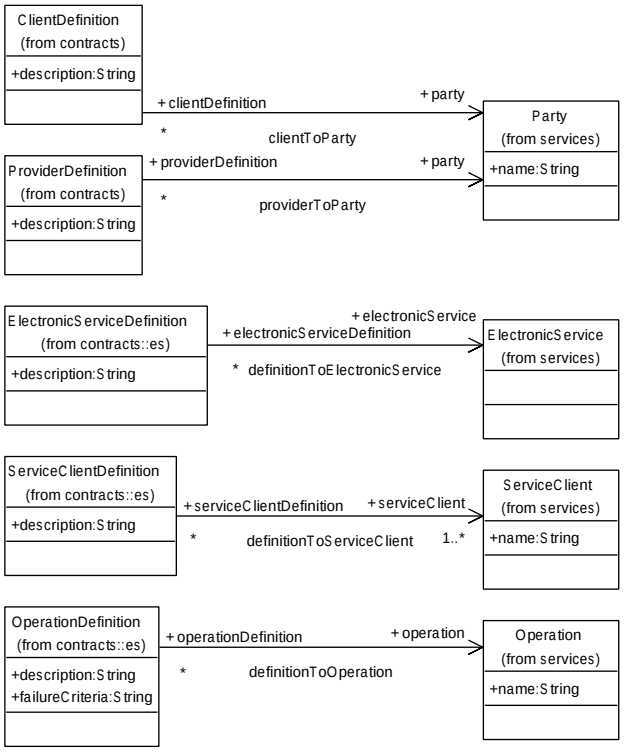

Fig. 3. The associations between syntax and semantic model elements defines the meaning of the language

alternatively be interpreted as a model of data describing the world, and the set of conditions necessary for those data to be considered free from violations. If we interpret the meta-model in this way, then we can produce a computer program capable of holding those data. The program can then check those data, to see if the world is in fact behaving in the way that we want it to, i.e. without violations of SLAs.

The process of implementing the checker program has the potential to introduce errors, such that the program either misses violations defined by the language specification, or reports violations that have not actually occurred. The second idea in this paper is that the potential for such errors can be substantially reduced by automatically generating the checker from the specification, rather than requiring human programmers to interpret the specification. The SLAng meta-model is ideally suited to this approach for the following reasons:

1) It is entirely expressed in a machine readable form. The meta-model itself is a MOF model, and may be represented in XMI. The constraints are in the textual format of the OCL.
2) A standard already exists for transforming MOF models into code, called the Java Metadata Interface (JMI) standard [13]. It defines a set of interfaces for manipulating models based on the structure of their meta-model.
3) A standard already exists for interpreting OCL constraints programmatically, and implementations have been produced.

Therefore, all that is necessary in order to implement a checker for SLAng SLAs is to generate the JMI interfaces and an implementation for the SLAng meta-model, and attach an OCL interpreter that can check constraints by querying these interfaces. This approach is shown in Figure 4.

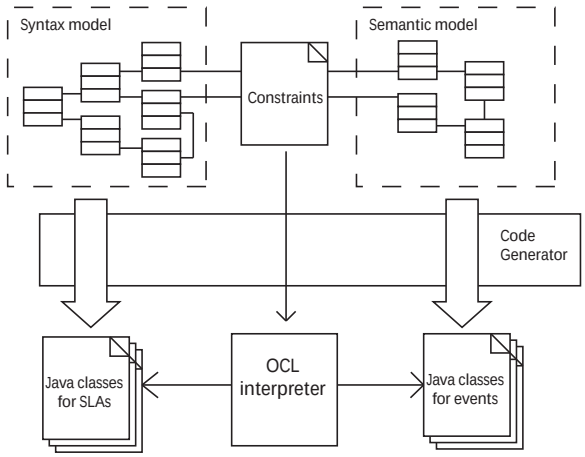

Fig. 4. Generating a contract checker from the SLAng metamodel

We achieved this goal by implementing a JMI generator. As discussed in the related work section, this was necessary because previous generators did not offer adequate flexibility over the type of code generated. We combined the resulting generated data structures with the OCL2 interpreter implemented at Kent University [14], 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 V.

IV. Design of the JMI generator

The JMI generator is implemented in Java, and follows the design shown in Figure 5. It is heavily dependent on the Velocity Template Engine (VTE) [15], developed as part of the Apache project. Similar to Java Server Pages (JSP) [16], or PHP [17], Velocity is a tool for generating text from predefined templates. These templates are text files, embedded in which are 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 the following Java types to be produced, each of which is contained in its own file:

- For each class:
  - A 'class proxy' interface, for creating and finding instances of the class.
  - An 'instance' interface, for editing properties and invoking operations of instances of the class.
- For each association: An 'association proxy' interface for creating and querying pairs of associated instances.

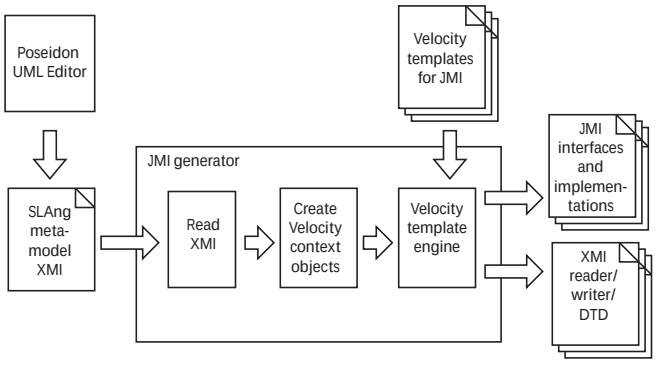

Fig. 5. Design of the JMI generator

- For each package: A ‘package proxy’ interface enabling the discovery of class proxies, association proxies and subpackage proxies.
- For each enumeration:
  - An interface type for enumeration values.
  - A class containing static exemplars of enumeration values.
- An XMI reader interface.
- An XMI writer interface.

The generator includes a template for each of these elements. Figure 6 shows the fragment of the template for the instance interface that generates accessor methods for attributes. Figure 7 shows the template applied to the context data for the ServiceUsage class shown in Figure 2. It extends a the more general event interface, adding methods for setting the ‘failed’ attribute (generate by the template shown in Figure 6, and references to the associated service client and operation.

```
## Accessor Operations
#*      *##if($a.multiValued)
    public static Class get${aNameCaps}_elementType =
        ${type}.class;

#*          *##if($a.ordered)
    public java.util.List get${aNameCaps}()
        throws javax.jmi.reflect.JmiException;
#*          *##else
    public java.util.Collection get${aNameCaps}()
        throws javax.jmi.reflect.JmiException;
#*          *##end
#*      *##else

    public $type get${aNameCaps}()
        throws javax.jmi.reflect.JmiException;
#*      *##end
## Mutator Operations
#*      *##if(!$a.multiValued && $a.changeable)

    public void set${aNameCaps}($type ${a.name}) throws
        javax.jmi.reflect.JmiException;
#*      *##end
#*  *##end
```

Fig. 6. Template for attribute methods on JMI instance interface

Except in the case of enumerations, the JMI specification only defines interfaces, but does not indicate how they are to be implemented. The generator therefore also includes templates for implementations of each of the above elements. Our simplistic implementation currently implements every

```
package uk.ac.ucl.cs.slang.model.services.es;

public interface ServiceUsage
        extends uk.ac.ucl.cs.slang.model.services.Period {

    // Attributes

    public boolean getFailed()
        throws javax.jmi.reflect.JmiException;

    public void setFailed(boolean failed) throws
        javax.jmi.reflect.JmiException;

    // References

    public uk.ac.ucl.cs.slang.model.services.ServiceClient
        getServiceClient()
        throws javax.jmi.reflect.JmiException;

    public void setServiceClient(
        uk.ac.ucl.cs.slang.model.services.ServiceClient
            newValue)
        throws javax.jmi.reflect.JmiException;

    public uk.ac.ucl.cs.slang.model.services.Operation
        getOperation()
        throws javax.jmi.reflect.JmiException;

    public void setOperation(
        uk.ac.ucl.cs.slang.model.services.Operation
            newValue)
        throws javax.jmi.reflect.JmiException;

    // Operations
};
```

Fig. 7. JMI interface to service usage data

proxy and instance as a separate Java object. All instances are stored in main memory simultaneously. The generator also has templates to implement the XMI reader and writer interfaces, and 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 Poseidon in an XMI format file. The first stage of the JMI generator reads this file and creates an in-memory representation of it.

In theory, the XMI reader for the UML models could be generated automatically from the UML metamodel, using a template derived from the XMI specification. Moreover, the loaded model should properly be manipulated using the JMI interfaces. Now that the JMI generator is implemented, we can generate these things. However, this is a chicken and egg situation, so the first stage is currently hand-implemented. The in-memory representation does not follow the JMI standard but is a simple data-structure reflecting the hierarchical structure of the XMI document.

The initial in-memory representation of the API is not suitable context for the Velocity templates. Velocity templates can perform only 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 therefore creates a number of different context objects, appro-

priate 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 filesystem.

V. The contract checker

A. Design

The contract 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.

The checker may be incorporated in electronic service systems wherever SLAs need to be monitored. Its use is 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. This is the moment when it is necessary to have a clear understanding of to what the terms in the agreement refer. The associations being established are shown in Figure 3. The links between the elements are created using the JMI interfaces.
5) Monitoring data is provided to the component by invoking the various 'create' methods found on the JMI API. This data is 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 Kent OCL implementation permits the evaluation of arbitrarily typed OCL expressions over the model (rather than being restricted to binary expressions – constraints). We have made use of this facility by associating a set of diagnostic expressions with each violation. If the constraint identifying the violation is found to have failed, then the diagnostics are evaluated to provide extra information as to the cause of the failure. For example, if the performance and reliability constraint shown in Section II is found to have been violated, then a diagnostic is used to calculate the observed reliability. These statistics are combined in a 'complaint' message, constructed using the Velocity template engine.

To demonstrate the contract 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. This is implemented using the reflective facilities of the JMI, which allows each element in a model to contain a link to its corresponding meta-element in its meta-model. The meta-model in this case is the MOF model instance representing the SLAng meta-model. It is stored in JMI classes generated (using the same JMI generator as described in the previous section) from the MOF model. The representation of the SLAng meta-model is only necessary when using the user-interface, and would not be required when using the checker as a component.

The user-interface also allows interactive editing and checking of the constraints over the SLAng model, and the diagnostics and complaint messages associated with violations of these constraints. The design of the checker is shown in Figure 8. A screenshot of the user interface is shown in Figure 9. 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.

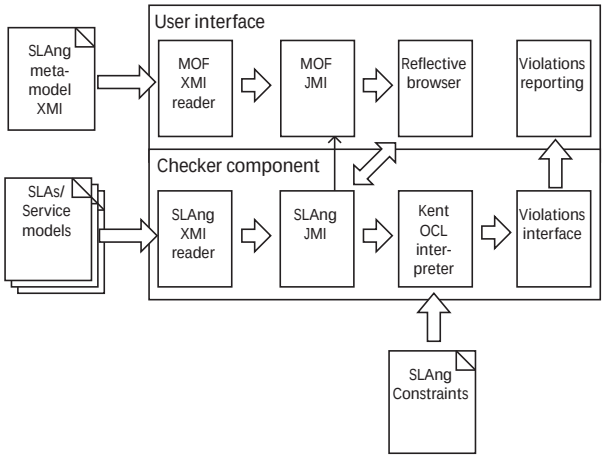

Fig. 8. Design of the SLA checker

B. Discussion

To establish that the contract checker respects the semantics of the SLAng language, we must in some way validate its correctness with respect to the language specification. It does not seem feasible to prove formally and exhaustively that the process of generating the JMI interfaces and implementations, or interpreting the OCL constraints introduces no such errors. However, the strength of the approach taken, and the main contribution of this paper is the observation that by generating the checker according to patterns (the JMI specification) that are standard and independent of the application domain (checking SLAs), then such errors are unlikely to be introduced, supporting our objective of making SLAng a more than usually precise SLA language.

This observation can receive some corroboration by testing our implementation. To date we have not conducted a thorough

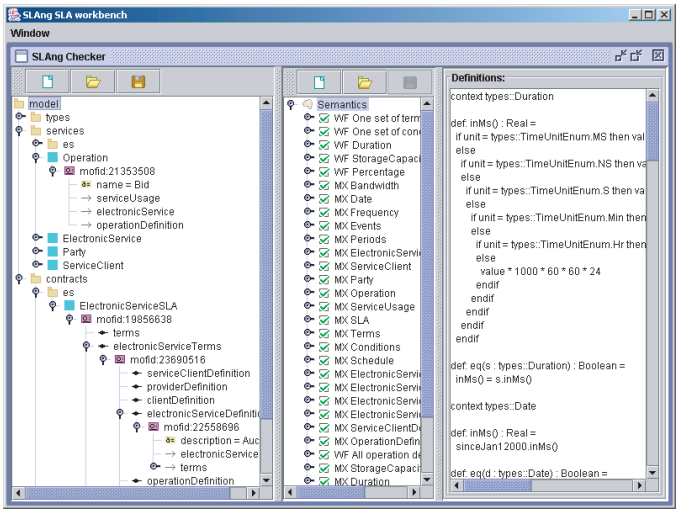

Fig. 9. Screenshot of the SLA checker user interface

and systematic test of the component, although what informal testing we have performed so far has tended to reveal errors in the language specification, rather than in the component's interpretation of it. However, a more thorough study is required in the future.

In order to function as a useful contract checker, the component must also be able to check for violations in a reasonable amount of time, over data sets of a realistic size. So far we have only tested our implementation on small models: that is, models containing few SLAs and little monitoring data. In these cases, violations can be checked for within a second or two. More formal characterisations of the performance of the component are required. However, there is reason to believe that the design as presented will be deficient in several respects:

The ASP SLAs are currently defined as implying constraints over a complete record of every service invocation. This data is likely to be very extensive, which will have implications both for the evaluation of the constraints and the management of performance data. The constraints as written are highly recursive and may require some optimisation to control their complexity. Moreover, in our current implementation of the component, service data is recorded in main memory and never deleted, leading to inevitable memory exhaustion.

Some of these problems can be solved be refining the implementation of the component, for example by providing a more sophisticated implementation of the JMI interfaces that relies on a database to persist service usage data, or by translating the OCL constraints into Java instead of interpreting them. Conversely, some problems will have to be solved by modifying the SLAng semantics. For example, if the volume of monitoring data renders either service provision or SLA checking infeasible, then the constraints will need to be redefined in terms of samples of monitoring data, rather than the total performance of the service.

At present we are preparing a large scale demonstration of the TAPAS project technologies, including SLAng contract checking, in the context of an auction house scenario. We expect that this experience will give us the opportunity to address the practical issues involved with checking SLAs.

VI. Related work

In [2] 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. We are not aware of any other attempt to automatically generate a checker for an SLA language.

However, our work does bear some resemblance to efforts to embed requirements monitors in software for runtime validation of systems. Systems for this purpose consist of a language for expressing the requirements, coupled with a mapping onto monitoring solutions. Representative examples are: the Java-MaC system [18] which automatically embeds monitors in Java code using a combination of bytecode rewriting and runtime libraries; and the KAOS-FLEA [19] 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.

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 from hand instead.

Probably the most commercially significant generator is the Eclipse Modelling Framework (EMF) [20]. The EMF generates specific repositories from UML meta-models according to a pattern similar to JMI. However, it is not template driven, so we would have no control over the implementation of the repository. If, as suggested in the previous section, we need to implement a repository backed by a database, it would be difficult to achieve using the EMF.

Another alternative is the AndroMDA tool [21], implemented using Velocity templates. The architecture of this tool is essentially identical to that presented in Section IV. Custom templates can be configured by the user, and the tool parses XMI representations of models and makes available standard context objects . However, as stated earlier, 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 outputs using AndroMDA. For example, the XML DTD 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 [22]. 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. For this reason we preferred to use more conventional templates.

In future we would like to see a combination between the template-based approach of AndroMDA, and the more powerful control structures available from OCL. One possibility is the use of PHP, a template language with sophisticated control structures. The use of PHP to generate code from models could be facilitated by providing a mapping of the MOF model to PHP to provide a standard interface to model data, comparable to the facilities provided by the JMI for Java.

VII. Conclusion

This paper has presented our approach to automatically generating an implementation of a contract checker from the specification of our SLA language, SLAng. We have argued that because the process of generating the checker is standard and independent of the semantics of SLAng, then semantic errors are less likely to be introduced into the checker. This allows the checker to be deployed and its results trusted with a greater degree of confidence by the parties to the agreement. The possibility of generating such a checker from the language specification enhances the utility of the specification considerably, and is a direct consequence of our adoption of a standardised modelling approach to describe the language and its effect in the domain of application. This work can be viewed as a case study, in a novel application area, of the application of MDA principles, including the definition of new high level languages, and the use of generative programming approaches to reduce the cost and increase the quality of software development projects.

Several practical and theoretical challenges remain to be addressed. The operation of the contract checker must be validated through testing. Further engineering needs to be applied to ensure that it can be deployed in realistic contexts. These efforts will be divided between improving the design of the checker (by modifying the way that it is generated), and improving the design of the language, so that its semantics do not imply too great a burden of monitoring. The language also requires expansion, to different kinds of services, including hosting and applications service provision, and modification to acknowledge that error is introduced by the process of monitoring, and that violations should therefore be associated with some degree of uncertainty.

Our initial implementation of the contract checker has served as a proof of concept, and also provides a useful test platform for refining future versions of the language, since the previously theoretical constraints and semantic models can now be tested against real and synthesised scenarios of service usage.

References

[1] 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*. IEEE Press, 2003, pp. 100 – 106.

[2] J. Skene and W. Emmerich, "Precise service level agreements," in *26th International Conference on Software Engineering (ICSE)*. Edinburgh, UK: IEEE Press, May 2004.

[3] *UML 2.0 OCL Final Adopted specification*, ptc/03-10-14 ed., The Object Management Group (OMG), October 2003.

[4] *MDA Guide Version 1.0.1*, omg/2003-06-01 ed., The Object Management Group (OMG), June 2003.

[5] J. Skene and D. D. Lamanna, "The SLAng Specification," 2003, http://www.cs.ucl.ac.uk/staff/j.skene/slang.

[6] *The Meta-Object Facility v1.4*, formal/2002-04-03 ed., The Object Management Group (OMG), April 2002.

[7] A. S. Evans and S. Kent, "Meta-modelling semantics of UML: the pUML approach," in *2nd International Conference on the Unified Modeling Language*, ser. Lecture Notes in Computer Science (LNCS), vol. 1723. Colorado, USA: Springer-Verlag, 1999, pp. 140 – 155.

[8] "The Object Management Group (OMG)," http://www.omg.org/.

[9] "Poseidon UML Editor," Gentleware A. B., http://www.gentleware.com/.

[10] *UML Profile for Meta Object Facility*, formal/04-02-06 ed., The Object Management Group (OMG), February 2004.

[11] *XML Metadata Interchange (XMI), v1.2*, formal/02-01-01 ed., The Object Management Group (OMG), January 2002.

[12] *EXtensible Markup Language (XML) 1.0 (Third Edition)*, The World Wide Web Consortium (W3C), February 2004, http://www.w3.org/TR/2004/REC-xml-20040204/.

[13] *Java(TM) Metadata Interface (JMI) API Specification 1.0 Final Release*, Java Community Process, June 2002, http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html.

[14] D. Akehurst, P. Linington, and O. Patrascoiu, "OCL 2.0: Implementing the Standard," Computer Laboratory, University of Kent, Tech. Rep., November 2003. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2003/1746

[15] "The velocity template engine v1.4," The Apache Jakarta Project, http://jakarta.apache.org/velocity/.

[16] "Java Server Pages JSP v. 2.0 specification," Sun Microsystems, Inc., http://java.sun.com/products/jsp/.

[17] "PHP: PHP Hypertext Preprocessor," http://www.php.net/.

[18] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-mac: a run-time assurance tool for java programs," in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu, Eds., vol. 55. Elsevier Science Publishers, 2001.

[19] 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*, 1998, pp. 50–59. [Online]. Available: citeseer.nj.nec.com/article/feather98reconciling.html

[20] "The Eclipse Modelling Framework (EMF)," The Eclipse Project, http://www.eclipse.org/emf/.

[21] "AndroMDA code generation tool," http://www.andromda.org/.

[22] "The Kent Modelling Framework (KMF)," The University of Kent, http://www.cs.kent.ac.uk/projects/kmf/documents.html.