Engineering an SLA Checker using MDA Technologies*

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), specified using technologies of the Model Driven Architecture approach (MDA), including the Meta-Object Facility model (MOF) and Object Constraint Language (OCL). In this paper we describe our motivation and experience in applying additional MDA technologies, including the Java Meta-data Interface (JMI) mapping and an OCL evaluator, to the automated generation of a software component for detecting SLA violations in service performance data. We highlight correctness and ease of implementation as particular benefits of the approach. These are significant as SLAs may form part of legal contracts and are expressed in a language that is subject to a degree of change. We include an evaluation of the component, employed to check the performance of an Enterprise JavaBeans (EJB) application. We discuss scalability issues resulting from immaturities in the applied technologies, leading to recommendations for their future development.

1 Introduction

In [16] 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.

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.

As described in [22] we have identified the need for SLAng to have rigorously defined semantics, to eliminate the possibility of parties disagreeing over the meaning of an SLA. We chose to apply 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 [17] (similar to a UML class diagram [20]). 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. SLAs may be associated with services, and Object Constraint Language (OCL) constraints embedded in the model assert that service behaviours should be consistent with the values specified in associated SLAs, hence defining the violation semantics for SLAng SLAs.

MOF and OCL are standards maintained by the Object Management Group (OMG), and are technical components of the emerging development strategy that it promotes called the Model Driven Architecture (MDA) [19]. In this approach, systems are developed by first modelling them in a technologically neutral manner, then refining models by adding platform-specific information, and finally deploying systems by automatically generating platform artefacts such as source code and deployment descriptors from models.

The fact that the SLAng language is described using MDA modelling language technologies presents the opportunity to apply the MDA approach of generating source code from models to generate the implementation of an SLA checker. The meta-model provides a specification of the data structures needed to store the pertinent SLA and service usage data, and the OCL constraints in the meta-model define what it means for this data to be considered free from violations. In this paper we describe how we have used the Java Metadata Interface (JMI) standard to generate

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

classes to store the data, and applied a free implementation of an OCL interpreter to interpret the OCL constraints from the SLAng meta-model over these data, and therefore detect violations if any exist.

This paper is an extended version of [21], adding an evaluation of the generated component employed to check the performance of an Enterprise JavaBeans (EJB) application running in the application server JBoss [8].

The main contributions of this paper are: firstly, to observe that the explicit meta-modelling approach used to define SLAng also effectively delivers the specification of a component for storing and interpreting data relevant to SLAs, eliminating the cost of reimplementing the checker when the language changes; secondly, to observe that by generating a component for interpreting a language automatically from the language specification we expect to reduce the chance of semantic errors being introduced during the implementation process; and thirdly to describe and evaluate our experience and the practical issues arising from taking this approach to producing the checker.

In outline, our paper reads as follows: In Section 2 we briefly review the features of the SLAng language and its specification. In Section 3 we describe in more detail the motivation for generating a checker component automatically, and the approach taken to achieve this. 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 EJB service, and present our evaluation. In Section 7 we discuss related work. Finally, in Section 8 we make some concluding remarks, and discuss future work.

2 Overview of the SLAng language

The SLAng language syntax and semantics are defined by a MOF (version 1.1) model [17]. 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 [12].

When SLAng was initially presented in [16] it could express SLAs for a range of different types of service including application service provision, component hosting, storage service provision and Internet service provision. However, since adopting the meta-modelling approach described in [22], we have only completed the meta-model for Electronic Service SLAs, which cover the provision of an application service over a network. The models and discussions in this paper therefore pertain to ES SLAs only. In future we intend to expand the formal definition for the other types of services listed above. The development of the SLA checker component was intended to assist with the development of the language by providing a platform for experimenting with different types of obligations, and for verifying that the meta-model constraints are both syntactically correct and appropriate.

MOF models are very similar to UML class models [20]. 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 use of a MOF meta-model to define the syntax of SLAng confers the advantages of the XML Metadata Interchange (XMI) [18] standard, a standard for serialising MOF-defined metadata. The XMI mapping of the SLAng syntactic model constitutes the concrete syntax of the language.

A fragment of a SLAng contract is shown below. It shows a `ServerPerformanceClause` that places constraints on the login operation of the EJB application described in Section 6, and forms part of the SLA used in the evaluation of the component. The attribute values of the clause, such as `maximumLatency` and `reliability` are defined using references to typed objects defined elsewhere in the SLA, identified using their `mofid`. At the end of the fragment, the `Duration` object associated with the `maximumLatency` attribute can be seen. The `ServerPerformanceClause` is also associated with an `OperationDefinition`, and a `Schedule`. These attributes and associations of the `ServerPerformanceClause` can be seen to correspond to those expressed in the model in Figure 1.

```
<SLAng:ServerPerformanceClause
  xmi.id="mofid:4328595"
  name="Login performance"
  maximumLatency="mofid:12499840"
  reliability="mofid:19485920"
  maxTimeToRepair="mofid:517215">
  <SLAng:ServerPerformanceClause.operation>
    <SLAng:OperationDefinition
      xmi.id="mofid:12947963"
      description="Login performance measured
        at EJB container boundary"
      failureCriteria="Any exception">
      ...
    </SLAng:OperationDefinition>
  </SLAng:ServerPerformanceClause.operation>
  ...
  <SLAng:ScheduledClause.schedule>
```

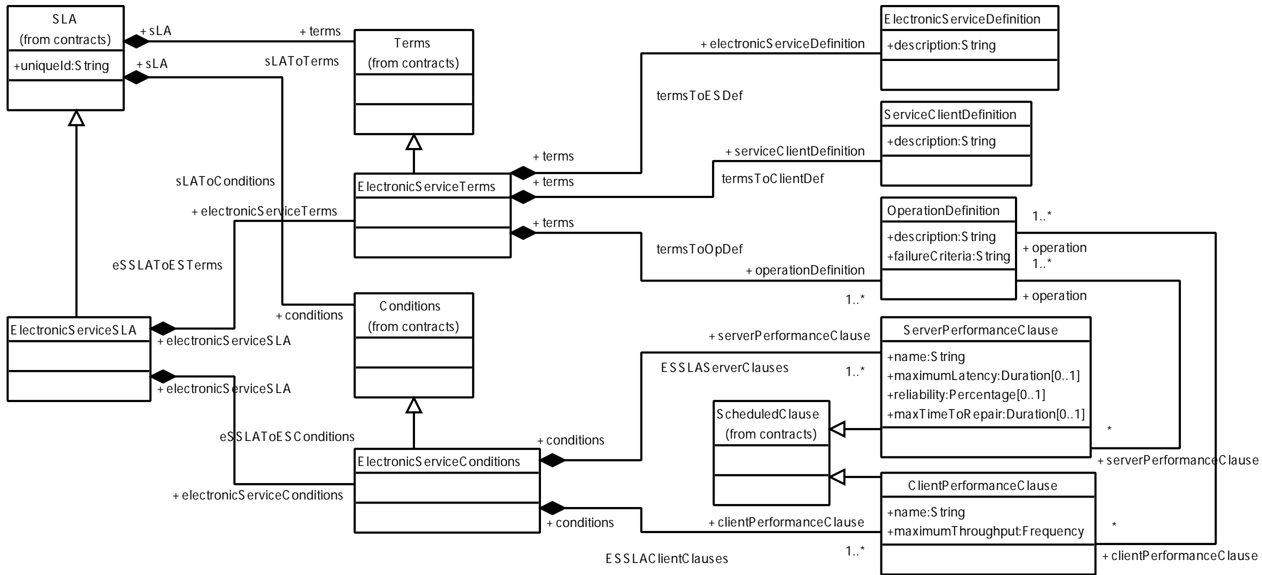

Figure 1. Model of the syntax of SLAng electronic-service contracts

```
    <SLAng:Schedule xmi.id="mofid:7361214"
      name="August to September 2004"
      startDate="mofid:8609150"
      duration="mofid:23958818"
      period="mofid:22674777"
      endDate="mofid:7286463"/>
  </SLAng:ScheduledClause.schedule>
</SLAng:ServerPerformanceClause>
...
<SLAng:Duration xmi.id="mofid:12499840"
  value="100.0"
  unit="mS"/>
```

The semantic model of electronic service provision is shown in Figure 2. 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. 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 Electronic Service SLA. 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) $\star$ (1 - reliability)))

This expression is explained in detail in [22][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.

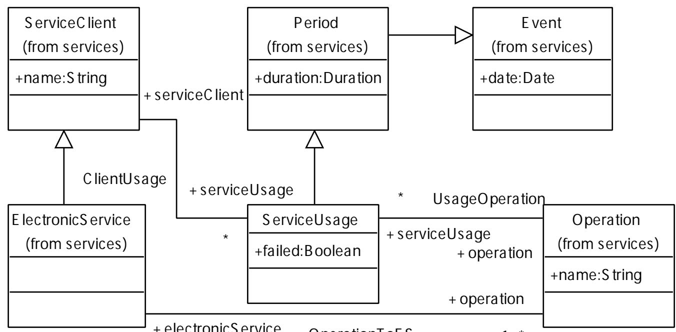

Figure 2. Model of electronic service usage

In contrast to other SLA and policy languages, SLAng does not include any intrinsic extensibility mechanisms, such as the capacity to define new sources of service performance data or composite obligations regarding the performance of services. It is our belief that languages providing these facilities without insisting on a strong supporting semantic definition for them pose a risk to the parties to the

[1]The expression is slightly modified from [22] 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.

SLA, as it is too easy to define ambiguous contracts, and in fact hard to define unambiguous ones (consider the 50 line OCL definition of performance and reliability discussed above). Instead we aim to provide useful and unambiguous contracts as the core definition of SLAng, and suggest that SLAng can be extended by modifying the meta-model and defining new constraints relating syntax to service behaviour, if necessary, and then with care. Of course, modifying the language necessitates the modification of SLA checkers, and this further motivates the need for a checker to be automatically derived from the language specification.

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 [22].

3 Generating an SLA checker

The SLAng meta-model and constraints, as used in the language specification, are a model of ideal service provision in the presence of SLAs. The model describes the structure of SLAs, and the structure and behaviour of services in the real world. The constraints assert that we expect the services to behave in a manner consistent with the SLAs that apply to them.

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 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 and checking them, to see whether services are behaving in the way that we want them 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. Moreover, every time the language is altered, during its development, or in response to changing requirements, checkers would require reimplementation. The cost of implementation and the potential for errors can be substantially reduced by automatically generating the checker from the specification. The SLAng meta-model is ideally suited to this approach: It is a MOF model, which may be represented in XMI, and the constraints are in the textual format of the OCL. It is therefore entirely machine readable. Moreover, a standard already exists for transforming MOF models into code, called the Java Metadata Interface (JMI) standard [14]. It defines a set of Java interfaces for manipulating models based on the structure of their meta-model. Finally, at least one implementation of an OCL interpreter is freely available.

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 3 in which thick arrows represent code generation, and thin arrows represent data flow.

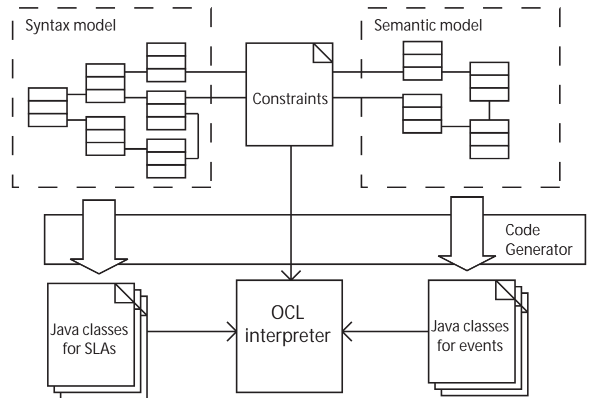

Figure 3. Generating an SLA checker from the SLAng meta-model

To achieve this goal we found it necessary to implement a JMI generator. As discussed in the related work section, this was needed 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 [11], 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.

4 Design of the 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) [10], developed as part of the Apache project. Similar to Java Server Pages (JSP) [5], or PHP [6], 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.

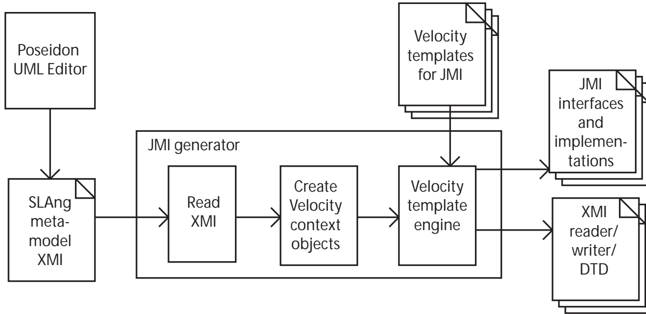

Figure 4. Design of the JMI generator

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.
- 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 5 shows a fragment of the template for the instance interface that generates accessor methods for attributes. Figure 6 shows the template applied to the context data for the `ServiceUsage` class shown in Figure 2.

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. The generator 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.

```
## 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
```

Figure 5. Template for attribute methods on JMI instance interface

```
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
};
```

Figure 6. JMI interface to service usage data

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.

5 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. This is the moment when it is necessary to have a clear understanding of to what the terms in the agreement refer. 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 (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 (Figure 2) provide the guidance as to what data these instruments must provide.

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. This relies on the reflective facilities of JMI, which allow 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. 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. The design of the checker is shown in Figure 7. A screenshot of the user interface is shown in Figure 8. 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.

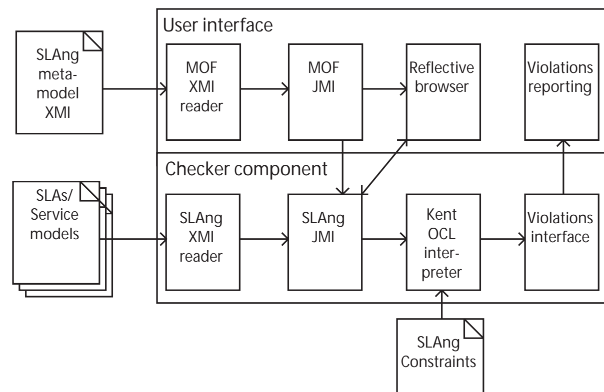

Figure 7. Design of the SLA checker

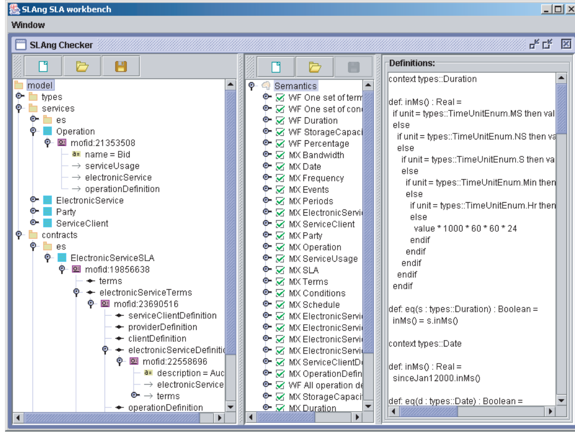

Figure 8. Screenshot of the SLA checker user interface

6 Evaluation

6.1 Deployment of the SLA checker

We have 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. SLAs are potentially very useful for auction applications, which typically involve multiple organisations, with mission-critical performance requirements. For the purposes of this evaluation we monitored the `login` operation using an SLA, a fragment of which is used as an example in Section 2. 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.

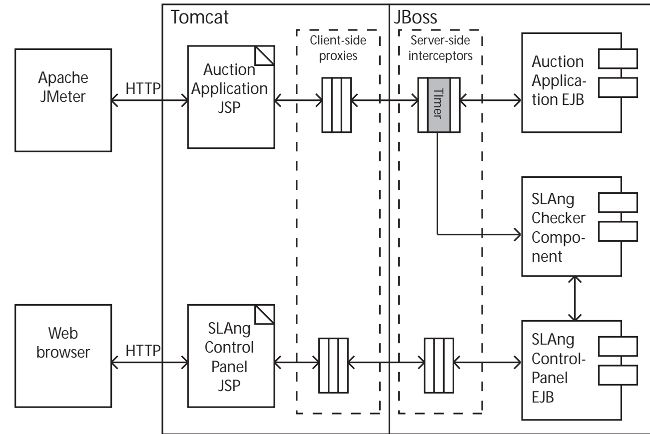

Figure 9. The SLA checker component deployed to monitor an EJB application

Service performance information is passed to the SLAng service by a server side interceptor configured as an option of the JBoss container configuration. 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 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 provi-

sion 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: One of the main claims of this paper is that by automatically generating the SLA checker from the language specification, errors in interpreting SLAs can be avoided. Our testing of the component has revealed many errors in the definition of the SLAng language, resulting from the fact that the original specification was developed without the assistance of an OCL interpreter. We also discovered several bugs in the OCL interpreter, although these caused it to conspicuously fail, rather than to return incorrect results. We have not yet detected any errors of the type mentioned above, and although we have yet to conclude a systematic testing of the component, we believe that this is encouraging.

However, 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. Moreover, 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. This seems an unacceptable bound on what is in essence a data processing application.

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 [22] 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.

Our implementation closely resembles that of the Kent OCL2 interpreter, that we employ to detect violations. Parts of the implementation of the OCL2 checker were generated from models of the OCL2 language syntax [11]. Moreover, its checks may be evaluated over models stored in Java classes generated by the Kent Modelling Framework, a code

generator similar to our own discussed below. In this sense the OCL2 interpreter uses automatically generated representations of both its syntax and semantics, and so is quite similar to the SLA checker. However, the OCL2 interpreter does not maintain any explicit representation of a large part of the language semantics, the process of interpretation.

Our work also bears 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 [15] which automatically embeds monitors in Java code using a combination of bytecode rewriting and runtime libraries; and the KAOS-FLEA [13] 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 by hand instead.

Probably the most commercially significant generator is the Eclipse Modelling Framework (EMF) [7]. 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 [1], implemented using Velocity templates. The architecture of this tool is essentially identical to that presented in Section 4. 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 above, 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 XMI 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 [9]. 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 over 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 of 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 classes. This would provide a standard interface to model data, comparable to the facilities provided by the JMI for Java, effectively allowing PHP pages to load their own context model before generating code. The resulting PHP pages would be more reusable than the templates in our implementation, as they would not depend on external code to represent and preprocess the context data.

8 Conclusion

This paper has described our use of MDA technologies to producing an implementation of an SLA checker, automatically, from the specification of our SLA language, SLAng. The approach means that the SLA checker can be regenerated automatically whenever the language changes, and 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. In these two respects, our experience can be seen as supporting two claims of the MDA approach: reduced costs and increased quality due to a reduction in human error. Moreover, the approach taken seems particularly appropriate when generating a checker for SLAs in which legal considerations may mean that it is important that the results generated by the component are particularly free from error with respect to the specification of the language.

The possibility of generating such a checker from the language specification can also be seen as a justification for our original choice of an explicit meta-modelling approach to defining SLAng. Designers of other languages may wish to consider adopting the approach as it offers the possibility to generate all or part of an interpreter for a language automatically. Where an explicit representation of the semantic primitives of a language is practical, an OCL interpreter can

be employed to check that these semantic elements are consistent with statements in the language, which is effectively what the SLA checker does when checking for violations.

In the process of implementing the checker we evaluated several code generation tools. These are discussed in the related work section. We believe that a template based tool is the easiest to use when performing code generation from models, but that the template language used should be expressive enough to allow preprocessing of the model data to be expressed in the template. We have proposed PHP as a possible suitable technology for future use.

Our evaluation of the checker revealed some serious practical considerations. In the case of the SLA checker, our in-memory representation of SLAs and service data takes the place of, and is in several respects subject to the same requirements as a database. OCL can be seen as acting as a query language over the data. 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.

Our initial implementation of the SLA checker has served as a proof of concept and as an opportunity to evaluate the technologies employed. It 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. Our future priorities will be to increase the sophistication of both the SLAng language and its SLA checker with the aim of producing a broadly applicable, precise language that can be used cheaply and correctly in realistic situations.[2]

References

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

[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] PHP: PHP Hypertext Preprocessor. `http://www.php.net/`.

[7] The Eclipse Modelling Framework (EMF). `http://www.eclipse.org/emf/`.

[8] The JBoss application server. `http://www.jboss.org/`.

[9] The Kent Modelling Framework (KMF). `http://www.cs.kent.ac.uk/projects/kmf/documents.html`.

[10] The Velocity Template Engine v1.4. `http://jakarta.apache.org/velocity/`.

[11] D. Akehurst, P. Linington, and O. Patrascoiu. OCL 2.0: Implementing the Standard. Technical report, Computer Laboratory, University of Kent, November 2003.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

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

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

[20] The Object Management Group (OMG). *The Unified Modelling Language v1.5*, formal/2003-03-01 edition, March 2003.

[21] 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, Monterey, California*. IEEE Computer Society Press, 2004. To appear.

[22] 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.

[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.