Automating SLA Modeling

Tony Chau, Vinod Muthusamy*, Hans-Arno Jacobsen*,

Elena Litani, Allen Chan, Phil Coulthard

IBM Canada Ltd.

University of Toronto*

Abstract

Service Level Agreements (SLAs) define the level of service that a service provider must deliver. An SLA is a contract between service provider and consumer, and includes appropriate actions to be taken upon violation of the contractual obligations. However, implementing an SLA using existing IT infrastructure is difficult, requiring a lot of manual effort to translate an SLA into code, model it with the given programming language, and ensure the required monitoring support is available for efficient monitoring and tracking of the SLAs.

In this paper, we present a solution for modeling an SLA contract. It is designed to be configurable, reusable, extensible and inheritable, thus providing great flexibility to construct complex SLAs. We also introduce an algorithmic generation pattern to create the necessary artifacts to implement an SLA presented in this paper. The resulting artifacts automatically monitor a business process and evaluate whether the SLA is violated during runtime execution. The proposed approach is designed to require minimal human intervention.

1 Introduction

The ubiquitous availability of Internet connectivity has substantially transformed how businesses are run today. A lot of companies automate their business processes and make them accessible through the Internet as Web services that other parties can subscribe to and integrate into their own business processes, in which the service is invoked on demand [1]. For example, a bank typically does not have the resource to record the entire credit history of individuals. In order for the bank to determine customers' credit rating, it subscribes to a credit check service from a third party and integrates the service into its own loan approval business processes. As services are purchased from different parties, it is always desirable to ensure that these services provide the expected quality of service as agreed upon by both parties. These qualities of service are more generally expressed in SLA contracts.

An SLA can be viewed as a contract between service consumers and providers. It defines the quality of service that both parties agree to consume and deliver. Optionally, an SLA also specifies the appropriate actions taken if the terms are violated; for example, charging the service provider as a penalty for the violation.

To ensure no party violates an SLA, the business process must be constantly monitored to report any violations. However, as more services are automated and accessible online, the task of such monitoring becomes more complicated, and manual monitoring becomes less feasible. In addi-

Paper appears in CASCON 2008, Richmond Hill, Ontario, Canada, October, 2008.

tion, implementing an automated monitoring service or process requires a lot of expertise and manpower; mainly due to the fact that no tool is available that facilitates this task. As a result, developers are required to assemble various technologies to deliver a monitoring solution, a process that often requires a lot of effort and expertise. Furthermore, the resulting solution is non-standard, often ad hoc, error prone, and not optimized for performance.

In this paper, we develop an approach to simplify this SLA development process. We first present a solution for modeling an SLA contract. The model is loosely coupled with the referencing business process allowing developers to evolve both the SLA and the business process implementation independently, without the need to regularly synchronize between the two artifacts. Furthermore, our solutions strive for high modularity and extensibility allowing SLA contracts to be constructed by assembling several elementary SLAs.

In addition, this paper also describes a runtime architecture that enables an effective execution of an SLA. This involves a validation framework that ensures the SLA is applicable to the referencing business process and the automatically generated monitoring artifacts that are deployed as a monitoring service. During execution, these artifacts monitor the referencing service and perform predefined actions upon violation of an SLA.

The rest of the paper is structured as follows: In section 2, background material on business process modeling languages is discussed. Section 3 describes the problem of SLA modeling and presents an overview of the proposed solution. Section 4 gives a related work survey on the examined topic with emphasis on similar approaches that are discussed in the literature. Sections 5 and 6 present the details of the proposed solution, which includes the SLA model and runtime architecture. In section 7, an end-to-end scenario is developed to illustrate the effectiveness of our solution. Finally, concluding remarks are given in section 8.

2 Background

Business processes can be automated in a number of different ways. Several open standards are available for modeling the logic of business processes, one of which is the Business Process Execution Language (BPEL). BPEL describes the business logic of a process, and defines the actual behavior of a business participant in a business interaction, without specifying the actual implementation [2]. Other business process modeling languages are also available, such as the Business Process Modeling Notation (BPMN) [3] and Web Services Flow Language (WSFL) [4], which both serve the same purpose as BPEL.

Every business process modeling language defines a set of basic constructs to be used for the specification of a business process. For instance, BPEL defines constructs such as the <receive>, <invoke> and <reply> activities to specify a business process. As an example, a loan approval process can be modeled using these constructs shown in figure 1.

```
<receive name="ReceiveApplication"/>
<invoke name="RetrieveCreditHistory"/>
<switch name="IsRatingGood">
    <case condition="rating='good'">
        <invoke name="Approve"/>
    </case>
    <case condition="rating='bad'">
        < invoke name="Reject"/>
    </case>
</switch>
<reply name="ReplyApplicant"/>
```

Figure 1: Loan Approval Process

The process is represented as a series of activities. The process begins with a <receive> activity that represent the arrival of a loan application. The process continues by retrieving the credit history of the applicant, which is accomplished by the first <invoke> activity in the process. If the applicant has an excellent credit rating, the request is automatically approved. Otherwise, it is sent to an official to process manually. In the example process, a <switch> decision activity performs this evaluation. The process then executes the corresponding <invoke> activity depending on the result of the credit check.

Once a business process is modeled, the process owner is free to choose the implementation of the services used in the process. In the above process, it makes use of the credit check service, which can be either implemented as an internal procedural call, or provided by a third party such as a credit rating agency. By combining different services together, a business process can be formed to handle complex business logic [5].

If a third party service is used, it is often desirable that both parties agree on the satisfactory level of service that the provider must fulfill and the service consumer expects. A Service Level Agreement (SLA) is introduced to address this requirement. It defines the quality of service that must be delivered by the service provider. The service consumer in turn relies on this level of service, as the service consumer might itself offer service guarantees to its consumers. The SLA also specifies the appropriate actions that must be taken if the terms and conditions are violated, such as the incurring of a penalty, for example.

There are a number of open standards available to model an SLA. For example, WSLA addresses the specification of SLA in a Web services environment [6]. WSLA defines the basic building blocks to model an SLA for Web services. In the specification, each SLA references the Web service as a whole. However, it can also refer compositions of multiple Web services [7]. It is composed of a number of metrics, which measure different aspects of a service. In the above example, the owner of the loan approval application might impose an SLA that every transaction must be completed in less than ten seconds. To enforce the SLA, metrics are created to measure the start and end time of the transaction. Metrics can also be defined by assembling simpler metrics to measure complexes scenarios. For example, after the start and end times are recorded, a metric for measuring the total transaction time can make use of the above metrics to compute the process duration.

The goal of the SLA is defined as Service Level Objective (SLO). It is a Boolean function expressed in terms of metrics. In the above example, the goal of the SLA on transaction time can be represented as an SLO in terms of the transaction time metric, in which the metric is compared with the agreed threshold. If an action must be executed when the SLO is violated, WSLA defines an Action Guarantee to specify the required actions.

In most cases, an SLA is verified during execution of a business process. However, it is also possible for an SLA to be verified at other stages. For example, an SLA to ensure that the service implementation is reviewed by a software architect should be verified during process deployment, since the requirement specifies that the software architect reviews the process before it is deployed for production.

3 Problem Overview

Realizing a given SLA is challenging, because today there exists insufficient tool support to address the issue of implementing the SLA for a given business process. The developer must make use of various technologies and compose them in non-standard ways to create a solution. The creation of the solution in this fashion is time-consuming and error-prone. In addition, the solution might not be optimized for performance and memory use. In order to implement the SLA, the developer must first analyze the SLA documentation in order to understand its requirements; for example, he or she must extract from the document the set of key performance indicators that govern the SLA. Once these indicators are identified, he or she must then instrument the business processes so that they can be measured. This is not trivial as the business process might require modifications to enable instrumentation. Moreover, additional programming logic might be required in order to measure performance of the process used to assess SLOs. Consequently, as discussed in [8], the process might have to be substantially modified to monitor the required SLA. This is usually not desirable because it discourages the reuse of the SLA and is difficult to extend and maintain both the SLA and the process.

If the process is capable of emitting events, the SLA developer can potentially separate the SLA logic into another business process. This new process is invoked when an interested event is emitted, and measures the indicators upon the arrival of events and thus evaluates the SLA. However, determining the set of events that is critical to SLA execution can itself be time-consuming, and the SLA developer typically simply enables all events to be emitted from the process in order to simplify the development task. Unfortunately, this inevitably increases performance overhead as unnecessary events are being emitted and processed.

To address this problem, this paper develops a methodology supported by an automated approach to determine the events required to evaluate an existing SLA and to determine the monitoring rules required to dynamically track SLOs checking for SLA violations. An overview of the solution is illustrated in figure 2. In the solution, it is assumed that a business process is a directed graph with nodes representing tasks or activities in the process and directed edges de-

scribing the control flow. It is also assumed that each activity in the process is assigned a unique identifier.

The solution involves first defining an SLA specified as prescribed by the approach defined in this paper, which is discussed later in section 4. In the approach an SLA is programmatically captured in a formal SLA model. The SLA model is loosely coupled with business process, so that developers can evolve both artifacts concurrently.

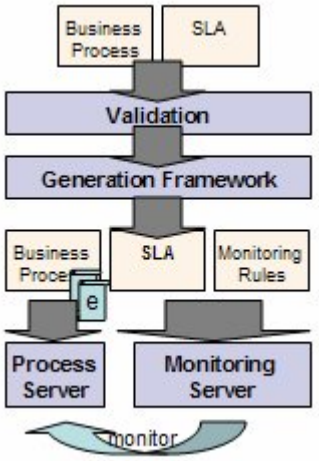

Figure 2: Overview of Proposed Solution

As shown in the figure, when both business process and SLA are complete, they are validated to ensure the SLA is still applicable to the business process. This is required because the established relationship might be broken during parallel development of the two artifacts. Subsequently, they are taken as inputs to the generation procedure.

The generation process involves identifying the set of events in the business process that are critical to the monitoring of the SLA. The occurrence of these events potentially signals the violation of the SLA. It is assumed that the business process is capable of emitting events when it enters and exits an activity, or when an exception is thrown. The event contains a snapshot of the current state of the process, so that important data can be retrieved from the event for evaluating the SLA. Examples of important data present in an event include unique identifier of the affected activity, input parameters, output parameters, and global parameters of the process. Once the events are discovered, the generation process ensures that the business process is configured to emit the minimal set of events required to monitor the given SLA for the given business process.

In addition to enabling the right events to be emitted by the business process, the generation process also creates a set of monitoring rules from the SLA. These rules are evaluated when an event of interest is emitted from the process. If any rule is violated, the appropriate action is triggered. Once the process is deployed in a server, the monitoring rules corresponding to the SLA are also deployed. These rules continuously monitor the process to enforce the SLA.

4 Related Work

There are several approaches in the design and development of automating an SLA for a business process. This section presents the most relevant ones.

Some researchers have focused on programmatically modeling an SLA. Sahai et al [9] first attempted to derive a set of constructs for modeling an SLA for Web services. In the proposal, elements such as SLA parameters and SLOs are introduced to model metrics and objectives respectively. Although some elements essential to SLA modeling are covered in this paper, other elements such as actions are not addressed in the proposed model. As a result, SLA developers are not able to execute an action upon SLA violation.

Keller and Ludwig [10] proposed different schematics for modeling SLAs. It is based on XML, which claims to provide some degree of extensibility. For instance, complex metrics are composed by a number of other metrics. In addition, SLOs can thus be defined in terms of metrics. This model poses some similarities with our proposed model. However, our model is more extensible and flexible with the introduction of metric and SLO libraries. For example, new metrics are created by extending a metric type. New metric types can also be created in the same fashion. The hierarchical relationship between metric and SLO types allows a maximum degree of extensibility and reuse.

On the other hand, Lamana et al [11] took a different approach in modeling SLA with the introduction of SLAng. It is an extension to existing business process languages. In SLAng, SLAs are defined in terms of a set of Quality of Service (QoS) parameters. These parameters are assigned to the target business process when it is being implemented. Executing these SLAs requires the target server to support these particular QoS parameters. The architecture becomes less extensi-

ble and flexible if new SLAs are introduced, because the server must be redesigned to support new QoS parameters.

A similar problem exists if the target SLA is implemented by a process. The runtime architecture described in [12] involves creating monitoring agents to monitor the target business process. These agents instrument the business process by listening to its network usage. When it detects a change in the process, it executes another process which evaluates the SLA.

The above architecture requires the business process to update constantly to adapt to new SLA requirements. In addition, these monitoring agents pose huge performance overhead, because they actively listen to every event emitted from the business process. As a result, events that are not critical to the SLA execution are also being processed by the system. It is different than our system as our system only accepts and processes events that are relevant to the SLA. Thus, our system consumes fewer computing resources. In addition, our system distributes the evaluation of an SLA among a set of monitoring artifacts, which can theoretically be run in a distributed environment. Its performance is generally better than the proposal in [12], in which the SLA is executed by a centralized business process.

The work in this paper is part of the eQoSystem project [13], which seeks to simplify the development and management of business processes through the use of SLAs. This includes automating the modeling and monitoring of SLAs as described in this paper, techniques to perform declarative service selection and composition [14], a distributed business process execution architecture [15], and algorithms to redeploy an executing business process without interrupting it [16].

5 SLA Model Architecture

Section 3.1 presented an overview of the proposed solution consisting of two major components: the SLA model and its corresponding runtime architecture. This section first presents the SLA model, and section 6 focuses on the runtime architecture that validates and generates runtime artifacts for execution.

As discussed previously, it is assumed that during execution the business process is capable of emitting events contain a snapshot of the current state of the business process. It is also assumed that the business process can be modeled as a directed graph with control flows that describe the business logic. These assumptions are essential to the proposed Service Level Agreement (SLA) model which extends the concepts from WSLA with some refinements. These refinements simplify the existing model without losing any critical information in defining an SLA. The SLA model is extensible and inheritable to allow maximum reuse. Complex SLAs can be constructed by composing a number of simpler SLAs and their components. Figure 3 shows the overall structure of the proposed model. Interested readers can also find its schema in Appendix A.

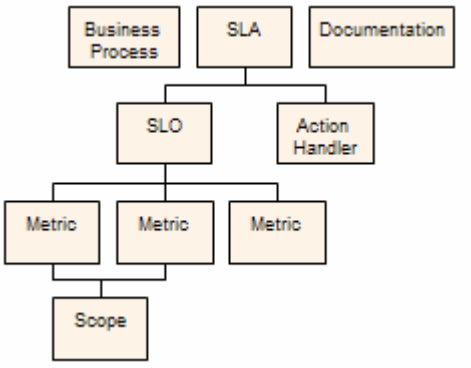

Figure 3: Overview of SLA Model

Each model has a reference to a business process being monitored. The reference is simply a file location of the business process definition, which can be expressed in any supported process modeling language. The model also includes a reference to the documentation detailing the contract on the level of service agreed by both parties. It is usually used by SLA developers as a reference during SLA modeling.

SLAs are the topmost elements of the model. As shown in figure 3, hierarchical relationships are established between SLAs and other components in the model. Top level components have dependency relationships with the components below. In addition, components are reusable so that multiple components can establish a dependency with the same component. In subsequent sections, each component is discussed in detail.

5.1 Scopes

A scope defines a region or subsection of the business process being monitored. The SLA model allows multiple scopes to be defined. A

scope is expressed as a series of start and end node pairs in the process diagram. For example, figure 4 shows a process diagram in which each activity is identified by a unique id. The figure contains two sections that overlap each other. The left region can be expressed as {1, 4} because it begins at node 1 and ends at node 4. This notation describes all path entries that begin with node 1 and end with node 4. Thus, paths 1-2-4 and 1-3-4 are considered to be in this region.

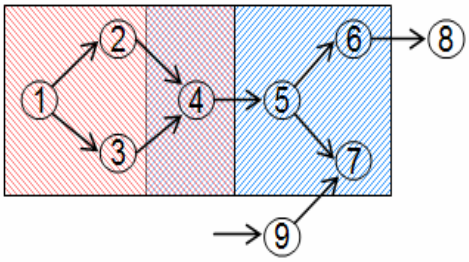

Figure 4: Examples of Scope

The proposed scope definition is flexible enough to express a wide variety of regions. For example, to express the scope on the right, we can define as a series of start and end node pairs, such as {{4, 6}, {4, 7}}. The sequence captures all path entries within this region. More complex regions can be expressed in a similar fashion.

5.2 Metrics

Metrics play an important role in the proposed SLA model. It defines an indicator for measuring different aspects of a process. Not only does a metric measure the performance of a process, but it also captures other information that is critical to the evaluation of the SLA. For example, number of usage by all customers might be measured to evaluate whether the subscribed service is underused. Once the measurements are taken by metrics, one can easily construct a Service Level Objective (SLO) by assembling the metrics together.

In the proposal, a metric is expressed as an instance of a metric type in a metric type library. A metric type library is a collection of reusable metric types which users can augment with their own metric types. A metric type defines the specific data and format to be captured as well as the methodology and business logic in order to measure this piece of information. In addition, it specifies the set of events that triggers a measurement to be taken. In general, a metric type consists of the following components:

Type Identifier: A unique identifier is assigned to every metric type in the library. Because of its uniqueness, a new metric can be extended from a metric type by referencing the type's identifier.

Parameters: A list of required parameters is defined for every metric type. Metric types make use of these parameters for its operation. The actual values of these parameters will be provided at a later time when a metric instance is defined. Detaching the parameter values from the schema gives more flexibility to users to customize the metrics. For example, a user can create multiple metrics extending from the same metric type. By assigning different parameter values to each metric, he is able to customize them to measure different aspect of the process.

Every parameter in the schema specifies its data type and optionally its range. Parameter types include scalars, scopes, individual activities and other metrics. The scope parameter can define the regions where the metric applies, thus measurement is only taken for the given scope. Some metric types do not specify a scope as a parameter, as the target region is implicitly defined. For instance, the metric type for measuring the transaction time of the process does not require a scope, because the scope is implicitly defined to be the entire process.

Metrics can also be included as parameter of other metric types. These metric types usually require other metrics to assist in taking measurements. For example, to measure the percentage of successful execution of a process, a metric might require other metrics that measure the total number of executions and the total number of successful executions. As a result, the resulting metric type might require these two metrics as parameters.

Dependent Events Function: This function captures a list of events that must be emitted by the process. These events are critical because they trigger the dependent metrics to take measurements. For instance, to measure the process execution time, the corresponding metric type must be notified of the entry and exit events of the process. Thus, the dependent events function for this metric type is defined as $\{A_{entry}, Z_{exit}\}$, where

A is the start activity and Z is the end activity of the business process.

The function is invoked in various places in the solution. It is first called during validation to collect a set of events that must be emitted by the referenced process. Validation then ensures the process is capable of emitting all of the events as required by the SLA. It is also invoked to retrieve the list of events and enable them in the referenced business process. More details will be given in section 6.

Event Handler: The event handler of a metric type contains the logic of how a measurement should be taken. It is executed when a relevant event, specified in the dependent events function, is emitted. The triggering event is passed as input to the handler. In general, it is a function call during runtime execution. The handler retrieves necessary data from the event for its processing.

5.3 Metric Type Example

To illustrate the concept of metric types and metrics, an example is given in this section. In the loan approval example process, assume that credit check services are purchased from a third party. Figure 5 shows the process details. The process checks whether the person's credit history is located in a local database and retrieves it locally. Otherwise, it retrieves the information from a remote database, which is a more expensive call. An SLA is agreed to ensure the percentage of remote database accesses is under a certain threshold.

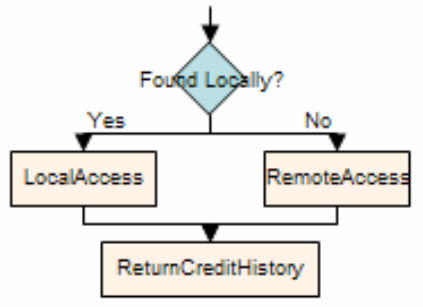

Figure 5: Credit Check Process

To measure the percentage of remote accesses, it is necessary to measure the number of invocations of both local and remote accesses. Thus, two metrics are required. However, because both metrics serve a similar purpose, they can actually be extended from the same metric type, which is shown in table 1.

Type Identifier	InvocationCountType
Parameters	Scope (type: scope)
Dependent Events Function	$\{Start_{entry}\}$

Event Handler	OnEvent(Event e) {
	static total;
	total= total+1;
	publish(total, e.instanceid);
	}

Table 1: Metric Type for Measuring Number of Invocations

This metric type takes a scope as a parameter. The scope identifies the area of interest. In this scenario, the two activities retrieve data from the database. To measure the number of invocations, the metric must be notified when the entry event of the scope occurs. When this happens, the event handler is invoked. The event handler updates the number of invocations and publishes an update event to notify others of the change.

Given the above metric type, the required metrics can thus be created to measure the number of invocations for both local and remote access. Tables 2 and 3 summarize the resulting metrics.

Name	LocalAccessCount
Metric Type	InvocationCountType
Parameters	Scope={LocalAccess}

Table 2: Metric for Measuring Number of Local Access

Name	RemoteAccessCount
Metric Type	InvocationCountType
Parameters	Scope={RemoteAccess}

Table 3: Metric for Measuring Number of Remote Access

The next step requires the measurement of the percentage of remote access. Table 4 shows the metric type that calculates the percentage. The percentage metric takes the above two metrics as input parameters and is notified when any of the input metrics emit an update event to indicate a value change. When the percentage metric is notified, the event handler is invoked to compute the new percentage. The percentage metric then fires an update event to notify others who are interested in the state change.

Type Identifier	CalcPercentageType
Parameters	Metric 1 {type: InvocationCount-Type } Metric 2 {type: InvocationCount-Type }
Dependent Events Function	{$Update_{metric}$}

Event Handler	OnEvent(Event e) {
	static percentage;
	percentage = metric 1 /
	(metric 1 + metric 2) * 100%
	publish(percentage, e.instanceid);
	}

Table 4: Metric Type for Computing Percentage

Given the above metric type, a new metric can be created, as shown in figure 5. The above two metrics are assigned to the new metric to measure the percentage of remote accesses.

Name	RemoteAccessPercentage
Metric Type	CalcPercentageType
Parameters	Metric 1= RemoteAccessCount Metric 2= LocalAccessCount

Table 5: Metric for Computing Percentage of Remote Access

5.4 Service Level Objectives

A Service Level Objective (SLO) defines a goal of the SLA. It is expressed as a Boolean expression in terms of metrics. The proposed SLA model allows multiple SLOs to be defined in one single model. Similar to metrics, SLOs are defined by extending an SLO type in the library. Continuing from the example in section 5.3, in order to define the SLO that the percentage of remote database access over total number of access is under a threshold, one can make use of the SLO type in table 6.

Type Identifier	LessThanSLOType
Parameters	Metric (type: CalcPercentageType) Threshold (type: int)
Dependent Events Function	{$Update_{metric}$}

Boolean Expression	OnEvent(Event e) {
	if(!(metric < threshold))
	publish(violate, e.instanceid);
	}

Table 6: Example of SLO Type

The above SLO type requires two parameters for evaluation: the metric that computes the percentage of remote accesses, and a scalar value that represents the desired threshold. If an SLO extends from the above SLO type, it will be triggered when an update event is emitted from the dependent metric, and its event handler is invoked to evaluate the condition. As shown in the table, a violation event is emitted when the condition is violated, so that interested parties are notified. This event captures the violated SLO which event recipients are able to retrieve. The resulting SLO is shown in table 7.

Name	CreditCheckSLO
SLO Type	LessThanSLOType
Parameters	Metric = RemoteAccessPercentage Threshold = 10%

Table 7: Example of SLO

5.5 Action Handlers

As mentioned earlier, when an SLO is violated, it is desirable to take appropriate actions upon it. For example, it might require emailing a manager, or charging the service provider a penalty. The proposed model addresses this requirement by action handlers.

Action handlers contain the logic that is executed upon a violation of an SLA. An action handle is typically a function written in any given programming language, and is invoked during process execution when a violation of an SLO is detected. To achieve a high degree of flexibility, the proposed model allows multiple action handlers to be defined within a single SLA model.

5.6 Service Level Agreements

As different aspects of an SLA are modeled in the proposed model, the SLA section is used to compose them together. An SLA is expressed as a set of pairs of SLOs and action handlers. The SLO specifies the goal of the SLA, whereas the action handler defines the action taken if the SLO is violated. To continue with the loan approval example, in order to model the SLA that ensures the percentage of remote access is under a predefined threshold, the SLA can be expressed as:

SLA = {PercentageSLO, chargeProvider}

Occasionally, several action handlers are required to execute upon a SLO violation. On the other hand, violations of multiple SLOs might cause the same action handler to run. For these reasons, the definition of SLA can be further gen-

eralized to encourage maximum reuse of SLOs and action handlers:

$$SLA = \{\{SLO_1 \ldots, SLO_n\}, \{Action_1 \ldots, Action_n\}\}$$

5.7 Discussion

The proposed model poses a number of advantages in modeling SLAs for any given process. First of all, the model itself defines metrics by extending metric types. By detaching the metric implementation as metric types, our model encourages reuse of metrics. As some metrics are commonly used by different processes, this architecture provides a great benefit to SLA developers.

Secondly, metric types make use of scopes and parameters to allow users to customize metrics to fit their needs. For example, by defining a metric type that measures the time taken for a scope, users are able to reuse it to measure different regions within a business process. Multiple metric instances can be created that extend the same metric type, each of which is then assigned to different scopes of interest. Another example can also be found in section 5.3, in which two metrics extend the same metric type to measure the number of invocation for different scopes. In addition, users can even further customize metrics by overwriting the default implementation of event handlers. It thus provides the capability to fine tune any metrics in the model.

Furthermore, the architecture also encourages complex metrics to be constructed by composing other metrics. Section 5.3 shows an example where a metric type consumes other metrics to compute the percentage of remote access over total access.

Similarly, the model architecture also encourages reuse of SLOs and action handlers. As SLO extend an SLO type, SLA developers can reuse the same type to create multiple SLOs. In addition, once an SLA is created, multiple SLAs can set it as their goal, which maximizes reusability of events further. Action handlers can also reused by multiple SLAs in the same fashion.

Because of the high reusability of the proposed model, it becomes easy to apply a SLA model from one process to another. For instance, to apply an SLA to another business process, one can update the business process definition to refer to the new process. As discussed later on in section 6, a validation will run to ensure that the SLA is applicable to the new process. It is possible that the scope definition is no longer valid, or the process itself is unable to emit events that are critical to the SLA execution. Any of these incompatibility issues will be indicated by validation, which requires user to correct them.

5.8 Implementation

The proposed SLA model has been implemented. A schema is written to capture the structure of the proposed model, which is attached in Appendix A. An editor allows users to create and modify the proposed model. The editor is developed and integrated into WebSphere® Integration Developer. A screenshot of the editor is shown in figure 6.

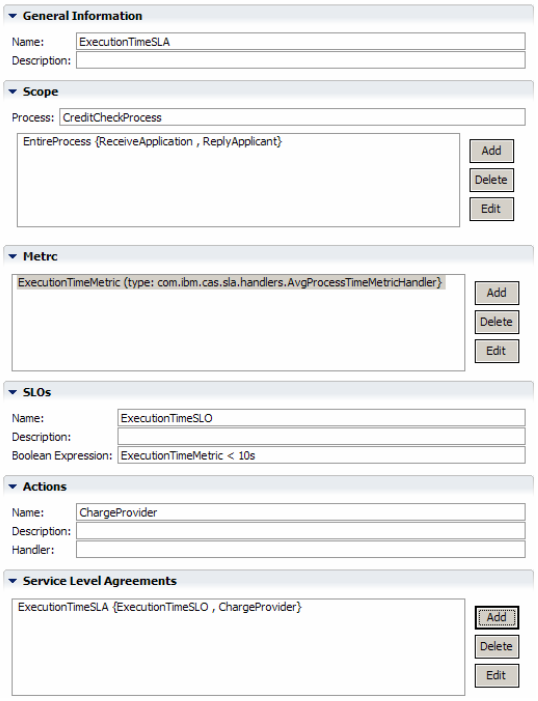

Figure 6: SLA Editor

Users are able to use the editor to develop an SLA model by creating metrics and SLOs from a type in the corresponding libraries. In addition, users are also able to create new metric and SLO types in these libraries through an extension framework, which provides the extensibility and flexibility capability introduced in the proposed model.

As for the next step of the implementation of the project, a wizard will be developed to generate monitoring artifacts given a completed SLA model and its referencing business process. This wizard is integrated with the backend of Web-

Sphere Monitoring Toolkit so that a monitoring model can be created which conforms to the SLA. Finally, the resulting monitoring model is then deployed in a WebSphere Business Monitor, which, during execution, monitors the referencing business process to detect violation of the SLA.

6 SLA Monitoring Architecture

The previous section discussed the details of the proposed SLA model. In this section, the architecture of executing the SLA is presented. As a first step, the SLA and its referencing process must be validated to ensure that the business process satisfies all requirements of the SLA; for example, all SLA scopes are resolved in the process.

After validation, the SLA and its referencing process are passed to the generation tool which generates the monitoring artifacts. It first derives the set of critical events used by the SLA and enables these events in the business process. The generation then generates a set of monitoring artifacts to execute the SLA. These artifacts respond to the events emitted by the process and thus evaluate the SLA.

To execute the generated monitoring artifacts, they must be deployed to a monitor server. Similarly, the referencing business process is deployed onto a process server. During execution of process, the monitoring artifacts monitor the process and ensure the SLA is not violated. In the following sections, details of the entire process will be discussed.

6.1 Validation

The validation process involves multiple steps. First, all scopes are validated to ensure they can be resolved in the process. In other words, the start and end activity of any scope must exist in the referencing process. Secondly, it also ensures that metrics are assigned with the correct set of parameters, as defined by their metric type.

Validation also analyzes the SLA and the business process to ensure the process is able to emit the set of events that are critical to SLA execution. The process begins by discovering all SLOs defined in the model. Starting from the SLOs, the validation process retrieves its dependent metrics from its hierarchical relationship. The process continues recursively until all nested metrics are obtained. For each metric, validation invokes its *dependent events function* to discover the set of critical events. The union of all resulting event sets is the set of events critical to the SLA execution. Given this event set, validation then ensures each event in the set can be emitted by the business process.

6.2 Event Enabling

As discussed earlier in this paper, the architecture listens to events emitted from the business process. It invokes the event handler of the affected SLAs to evaluate if the current process state violates the SLAs. Typically not all events are necessary for SLA monitoring. For example, an SLA to ensure the process execution time is under a limit requires only two events to be emitted. They are the entry and the exit event of the referenced process. It is very costly to process each event during execution, as it consumes both computing and network resources. It is desirable to reduce the performance overhead by enabling only the set of the events that are critical to the SLA evaluation.

The same algorithm as discussed in section 6.2 is used to determine the set of critical events. In summary, all SLOs defined in the model are discoverd. The discovered SLOs are used to discover the dependent metrics. The process continues recursively until all elementary metrics are found. After that, the *dependent events function* is invoked to discover the set of critical events. The union of all resulting event sets is the set of events critical to the SLA execution. Given this event set, the generation tool enables them in the referenced process.

6.3 SLA Execution

In addition to event enabling, the generation tool also generates a set of monitoring artifacts for the given SLA. These artifacts, when executed, monitor the business process to ensure the SLA is not violated.

As discussed in section 5, the proposed SLA model is structured hierarchically. Top level components depend on components below. If the state of any child element is changed, it potentially has an impact on the elements above in the model. For example, in the credit check process example, when an entry event of the local access activity is

emitted, the corresponding metric must be notified to update the number of invocation. As a result of the update, other metrics that depend on this metric must also be updated. In this case, the metric that computes the percentage must be updated. As shown in figure 7, there is ripple effect that causes a reevaluation of the SLO. Notice that a pattern similar to the publish-subscribe model is followed. As a result, we suggest that the events be disseminated using a publish-subscribe infrastructure.

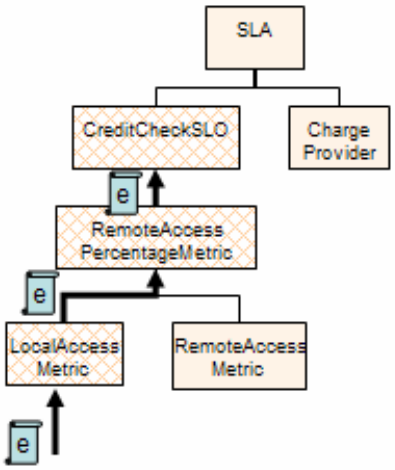

Figure 7: Event Propagation in SLA

Figure 7 shows the runtime architecture that executes SLAs. In the system, SLA components such as metrics, SLOs and action handlers become the clients of the publish-subscribe system. They all act as both a subscriber and publisher of events. To determine the set of events a client is interested in, its *dependent events function* is called to retrieve the list and subscribe to the relevant events.

To execute the SLA, the target business process is first registered as a publisher of the system. During execution, the process emits events previously enabled by the generation.

As shown in figure 8, when events from the business process enter the system (1), the publish-subscribe middleware forwards the events to interested clients. If the client is a metric, its event handler is invoked to update its value. The handler also emits new events into the system because of the update. These events are in turn forwarded to other interested clients (2), causing other clients in the system to reevaluate (3). This chain of actions continues until it reaches the top level clients (4). In this case, the action handler is executed because the corresponding SLO is violated.

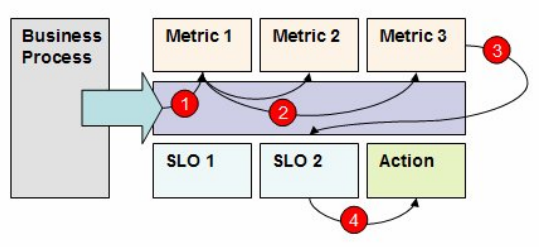

Figure 8: Overview of Runtime Architecture

This architecture poses a number of advantages. First, all clients in the publish-subscribe system are loosely coupled, which encourages reuse of these clients during runtime execution. For example, if several SLAs depend on same set of metrics, the monitoring server can reduce overhead by reusing its metric instances. It can easily be accomplished by switching program context when executing a metric

Furthermore, loose coupling also encourages SLA execution in a distributed environment. Since the architecture is highly modular, distributed computing can be easily achieved by moving these clients around the server cluster.

7 Case Study

In order to illustrate the proposed solution of modeling SLAs for a given business process, an end-to-end scenario is presented in this section. In this example, an insurance company models its claim approval process as shown in figure 9. The claim approval process is composed of a number of activities. The claim process starts off when an applicant submits a claim form. An automated process first evaluates the application. The application is automatically processed if enough information is provided and the applicant's credit history is good. Otherwise, the application is passed to a junior clerk to manually examine. Depending on the complexity of the application, he might escalate the request to his supervisor to handle. No matter who processes the application, an evaluation is given as a result. If the application is approved, the claim process continues with issuing a cheque to the applicant. Otherwise, the

process replies to the customer about the rejected reason.

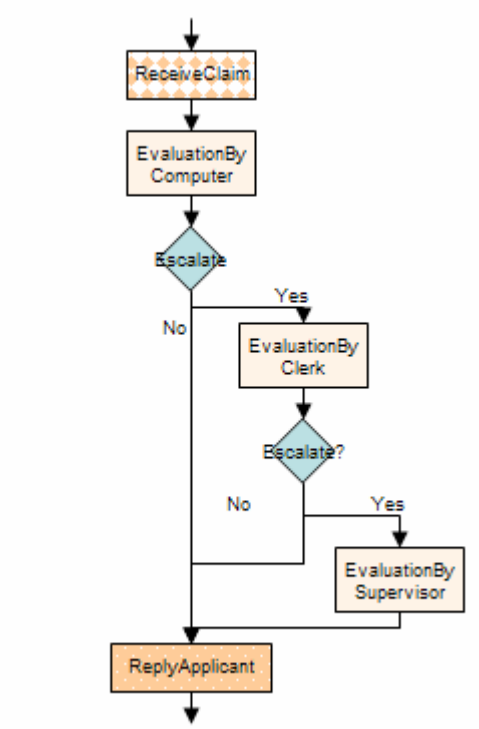

Figure 9: Claim Approval Process

The insurance company is interested in enforcing several SLAs to govern the given claim process. The company wants the total time taken by the manual evaluation to be less than 200 hours per day. In addition, because supervisors are expensive resources, the percentage of the time spent by all supervisors must be less than 20 percent of the total time taken by all persons. Also, the senior management team must be notified if any SLA is violated.

7.1 Modeling the SLAs

Given the above requirements, an SLA model can be created. To begin with, the SLA model first references to the claim approval business process. If a document is available that describes the SLA details, a reference to this document can be added in the SLA model.

7.1.1 Metrics

To continue modeling the required SLAs, the SLA developer must first identify the required metrics. Looking at the SLA requirements, it is known that three metrics are needed. The first metric measures the total execution time taken by all individuals, the second metric measures the total execution time taken by all supervisors only, and the last metric calculates the percentage by composing the first two metrics.

To define the first and second metrics, it is noticed that both metrics have similar nature in which both measure the execution time of a region of the process. As a result, it is thus preferable to define a metric type to measure the total execution time, as shown in table 8.

Type Identifier	TotalExecTimeType
Parameters	Scope (type: scope)
Dependent Events Function	$Start_{entry,}$ End_{exit} }

Event Handler	OnEvent(Event e) {
	static total = {i_1, …, i_n};
	static entry = {i_1, …, i_n};
	if(e is entry_event){
	entry[e.instanceid] = e.time
	}else{
	diff = e.time - entry[e.instanceid];
	total[e.date] = total[e.date] + diff;
	publish(total[e.date], e.instaceid);
	}

Table 8: Metric Type of Measuring Execution Time

The metric type measures the total execution time required by a given scope. It listens to the entry event of first activity of scope and the exit event of the last activity of scope to record the total execution time of the region. When an entry event is received, the event handler records the start time of the region. Similarly, the handler records the finish time when an exit event is received.

As discussed in this paper, it is assumed that the event captures a snapshot of the current state of the process; thus the start and finish time can be retrieved from the event itself. Once both times are recorded, the total execution time can be calculated and the result can be published to interested components. Given the above metric type, one can define the required two metrics as follows:

Name	TotalTimeByAllPersons
Metric Type	TotalExecTimeType
Parameters	Scope={{EvalutionByClerk}, {EvalutionBySupervisor}}

Table 9: Metric for Measuring Total Execution Time

Name	TotalTimeByAllSupervisors
Metric Type	TotalExecTimeType
Parameters	Scope={EvalutionBySupervisor}

Table 10: Metric for Measuring Execution Time of Supervisors

Table 9 shows the first metric which calculates the total execution time spent by all individuals. This metric extends from the metric type that is defined previously. Two scopes are assigned as input parameters when the metric is declared, so as to achieve the objective of measuring the total execution time spent by all individuals, which include junior clerks and supervisors. Similarly, the second metric extends the same metric.The second metric is assigned with one single scope which enables measuring the total execution time taken by all supervisors.

In addition to the above metrics, another metric is needed to calculate the percentage of the time spent by all supervisors over all individuals. A new metric type is introduced to serve the purpose. Table 11 shows the required metric type. It takes the above two metrics to calculate the percentage. Its value is updated by the event handler when an update event is emitted by any of the two metrics. If its value is updated, the event handler in turn emits an update event to notify interested parties.

Type Identifier	CalcPercentageType
Parameters	Metric 1 {type: TotalExecTimeType} Metric 2 {type: TotalExecTimeType}
Dependent Events Function	{$Update_{metric}$}

Event Handler	OnEvent(Event e) {
	static percentage;
	percentage = metric1 /
	metric2 * 100%
	publish(percentage, e.instaceid);
	}

Table 11: Metric Type for Computing Percentage

Name	PercentageTimeBy-Supervisors
Metric Type	CalcPercentageType
Parameters	Metric 1=TotalTimeByAllSupervisors Metric 2=TotalTimeByAllPersons

Table 12: Metric for Computing Percentage

7.1.2 Service Level Objectives

After all required metrics are defined; the SLA developer is able to declare the necessary SLOs. In this scenario, two SLOs are needed for the two SLAs being modeled. The first SLO ensures the total execution time is less than a threshold. Similarly, the second SLO also ensures the resulting percentage is less than a threshold. As a result, the following SLO type can be used to model both SLOs.

Type Identifier	LessThanSLOType
Parameters	Metric (type: Metric) Threshold (type: int)
Dependent Events Function	{$Update_{metric}$}

Boolean Expression	OnEvent(Event e) {
	if(!(metric < threshold))
	publish(violate, e.instaceid);
	}

Table 13: Example of SLO Type

The above SLO accepts a metric and a scalar to ensure that the given metric is less than the scalar value. Its condition is evaluated when an update event is emitted by its dependent metric. If the condition no longer holds, it emits a violation event, which contains a reference to the violated SLO. Given the above SLO type, the two required SLOs can be defined as follows.

Name	TotalTimeByAllPersonsSLO
SLO Type	LessThanSLOType
Parameters	Scope={TotalTimeByAllPersons} Threshold=100

Table 14: Example of Resulting SLO

Name	PercentageUnderTwentySLO
Metric Type	LessThanSLOType
Parameters	Scope={PercentageTimeBy-Supervisors} Threshold=20

Table 15: Example of Resulting SLO

7.1.3 Service Level Agreement

As mentioned earlier in the section, it is required the senior management team is notified if any SLA is violated. To accomplish the task, an action handler is written to inform the management team through e-mails. In this scenario, the action handler is implemented as follows.

```
onViolation (Event e) {
   if (e.violatedSLA ==
         TotalTimeByAllPersonsSLO) {
      sendTo("Joe Doe", e);
   } else {
      sendTo("Mary White", e);
   }
}
```

Figure 10: Implementation of Action Handler

The input of the handler is the violation event. It captures the current state of the business process. For example, it captures the SLO that is violated such that the handler is able to take appropriate action.

Once the action handler is implemented, the SLA developer can model the required SLAs as follows: The first SLA ensures that the total execution time used by all individuals is less than 100 hours per day, whereas the second SLA determines if the percentage of time spent by supervisors is less than 20 percent. If either SLA is violated, the management team will be notified by the action handler.

SLA1 = {TotalTimeByAllPersonsSLO, sendNotify}

SLA2 = {PercentageUnderTwentySLO, sendNotify}

7.2 Validation

To execute the SLAs above, a validation must be first executed to ensure the SLAs are applicable to the business process. First, all scopes defined in the model are validated. The start and end activity of every scope must exist in the process. Second, all metrics are extended from a metric type and they are assigned with the correct number of parameters.

As the final step, the validation retrieves a list of events that are critical to the SLA execution. It then ensures the process is capable of emitting these events. As discussed in section 5.1, the validation transverses the hierarchy of the model starting from the SLOs. A list of SLOs and metrics that are used in the model can thus be obtained. For each element, the *dependent events function* is invoked to retrieve the set of critical events. In this scenario, the claim process must be able to publish the entry and exit event of the supervisor human activity, as well as the entry and exit event of the clerk human activity.

7.3 Generation of Monitor Rules

After validation, the SLA and its business process are given to generate the set of monitoring artifacts. That is, the set of clients that will be registered in a publish-subscribe infrastructure.

The generation first retrieves the list of critical events and enables them in the business process. As a result, the business process, when executed, emits the required events to the infrastructure. Next, the generation generates a monitoring client for each metric, SLO and action handler in the model. In this scenario, a total of six clients are registered in the system, three of which are generated from the metrics, two of which from the SLOs and one from the action handler. These clients are then registered into the system. During registration, each client also subscribes the set of events as defined by its *dependant events function*.

8 Conclusions

This paper presents an approach to simplify the development of business processes governed by SLAs. An SLA model is proposed in this paper for effectively modeling an SLA. The model greatly extends WSLA which captures critical information of an SLA. The model is loosely coupled with the business process it is referencing. Developers can thus evolve both artifacts – business process and SLA model – concurrently. In addition, the proposed model is highly modular, which gives rise to extensibility to construct complex SLA contracts by composing elementary SLAs.

The paper also presents an architecture that supports execution of the proposed SLA model. It is developed based on a publish-subscribe system. In the design, each SLA is composed of a number of monitoring clients that coordinate with each other by exchanging events to verify if the SLA is being violated. The architecture is modular which encourages reuse of monitoring clients. It also provides great flexibility in the deployment of the SLAs. For example, the architecture is easy to distribute.

As shown in the paper, a graphical user interface has been developed to express and specify SLAs according to the model proposed in this paper. Currently, the proposed runtime architecture is being implemented using WebSphere products. To evaluate the proposed solution in

simplifying the SLA development, several approaches are being considered. The first approach measures the performance gain by using this solution. CPU consumption, memory and network usage are the key indicators for this evaluation. Other methodologies under consideration include the complexity of the solution in terms of lines of code, the time required, or the product skill set needed to model an SLA.

In the future, the solution will be further enhanced based on evaluation and feedback from users. One area that will be focused on is to extend the design to enable modeling of SLAs that apply to multiple processes and even across different processes. In addition, executing an SLA on a distributed system is another area that requires further research.

About the Authors

Tony Chau is a software engineer in IBM Canada Ltd. He began his career as a software engineer in 2001. He has been working in a number of WebSphere products, recently focuses on WebSphere Integration Developer. His e-mail address is tonychau@ca.ibm.com.

Vinod Muthusamy is a PhD candidate in the Middleware Systems Research Group in the Department of Electrical and Computer Engineering at the University of Toronto. His research interests include publish/subscribe systems and distributed workflow processing. Vinod holds a BASc degree from the University of Waterloo and an MASc degree from the University of Toronto. His e-mail address is vinod@eecg.utoronto.ca.

Hans-Arno Jacobsen holds the Bell University Laboratories Chair in Software, and he is a faculty member in the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto, where he leads the Middleware Systems Research Group. His principal areas of research include the design and the development of middleware systems, distributed systems, and information systems. Arno's current research focuses on distributed event-based processing and aspect-oriented software development.

Elena Litani is a Software Developer in IBM Toronto Lab. Elena has been at IBM for 8 years, working as a software developer, team lead and development manager on different projects, including Eclipse Modeling Framework (EMF) and Apache Xerces2 open source projects. She is currently a member of the Center for Advanced Studies at IBM. Elena holds a Bachelor's Degree in Computer Science from the York University (Toronto, Canada).

Allen Chan is the Lead Architect for BPM Connectivity Tools, responsible for the architecture and technical strategy for the ESB and Connectivity support in WebSphere Integration Developer and WebSphere Message Broker Toolkit.

Phil Coulthard is the development lead on WebSphere Integration Developer, which is the tool set for the WebSphere Process Server and WebSphere Enterprise Service Bus servers which build on WebSphere Application Server. Part of his team also works on the Message Broker Toolkit, which is the tool set for the WebSphere Message Broker server.

References

[1] A. Keller, G. Kar, H. Ludwig, A. Dan, and J.L. Hellerstein. Managing Dynamic Services : A Contract based Approach to a Conceptual Architecture. In R. Stadler and M. Ulema, editors, *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS 2002)*, pages 513-528, Florence, Italy, April 2002. IEEE Publishing.

[2] Business Process Execution Language for Web Services Version 1.1, BEA Systems, IBM Corporation, Microsoft Corporation, SAP AG, and Siebel Systems, (2002), developerWorks (updated February 1, 2005), http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[3] Business Process Modeling Notation Version 1.0, Business Process Management Initiative (BPMI), http://www.bpm.org/.

[4] F. Leymann. *Web Services Flow Language (WSFL) 1.0.* IBM Software Group, May 2001.

[5] R. Khalaf, N. Mukhi, and S. Weerawarana. Service-Oriented Composition in BPEL4WS, *Proceeding of the Twelfth International World Wide Conference (WWW2003), Web*

Services Track, Budapest, Hungary, May 20-24, 2003, Kluwer Academic Publishers, Norwell, MA (2003).

[6] H. Hudwig, A. Keller, A. Dan, R.P. King, and R. Frank, *Web Service Level Agreement (WSLA) Language Sepcification, Version 1.0*, IBM Corporation (January 2003), http://www.research.ibm.com/wsla.

[7] V.Tosic, B. Pagurek, B. Esfandiari, and K. Patel. Management of Compositions of E- and M-Business Web Services with multiple Classes of Service. In R. Stadler and M.Ulema, editors, *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS 2002)*, page 935-937, Florence, Italy, April 2002. IEEE Publishing.

[8] R. Khalaf, A. Keller, and F. Leymann. Business Processes for Web Services: Principles and Applications. *Celebrating 10 Years of XML*, Volume 45, Number 2, pages 425-446, January, 2006.

[9] A. Sahai, A. Durante, and V. Machiraju. Towards Automated SLA Management. HPL-2001-301.

[10] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services, *Journal of Network and Systems Management*, Volume 11, Number 1, pages 57-81, March, 2003.

[11] D. Lamanna, J. Skene, and W. Emmerich. SLAng: A Language for Defining Service Level Agreements. *Distributed Computing Systems, 2003. FTDCS 2003. Proceedings.* pages 100-106, May 2003.

[12] A. Sahai, V. Machiraju, M. Sayal, L. Jin, and F. Casati. Automated SLA Monitoring for Web Services. HPL-2002-191.

[13] V. Muthusamy, H.-A. Jacobsen, P. Coulthard, A. Chan, J. Waterhouse and Elena Litani. SLA-driven Business Process Management in SOA. *In Proceedings of CASCON 2007*, pages 264-267, October 2007.

[14] S. Hu, V. Muthusamy, G. Li and H.-A. Jacobsen. Distributed Automatic Service Composition in Large-Scale Systems. *In Proceedings of DEBS 2008*, pages 233-244, July 2008.

[15] G. Li, V. Muthusamy and H.-A. Jacobsen. Ninos: A Distributed Service Oriented Architecture for Business Process Execution. Middleware Systems Research Group Technical Report, July 2007.

[16] S. Hu, V. Muthusamy, G. Li and H.-A. Jacobsen. Transactional Mobility in Distributed Content-Based Publish/Subscribe Systems. Middleware Systems Research Group Technical Report, July 2007.

Trademarks

Appendix A: XSD Schema of SLA Model

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema
       xmlns:sla="http://www.cas.ibm.com/sla"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://www.cas.ibm.com/sla">
  <xsd:complexType name="DocumentRoot">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="scope" type="sla:Scope"/>
      <xsd:element name="action" type="sla:ActionHandler" maxOccurs="unbounded"
minOccurs="0"/>
      <xsd:element name="serviceLevelObjective"
type="sla:ServiceLevelObjective" maxOccurs="unbounded" minOccurs="0"/>
      <xsd:element name="metrics" type="sla:Metric" maxOccurs="unbounded" mi-
nOccurs="0"/>
      <xsd:element name="serviceLevelAgreements"
type="sla:ServiceLevelAgreement" maxOccurs="unbounded" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Scope">
    <xsd:sequence>
      <xsd:element name="process" type="xsd:string"/>
      <xsd:element name="ranges" type="sla:Range" maxOccurs="unbounded" minOc-
curs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Range">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="startNode" type="xsd:string"/>
      <xsd:element name="endNode" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Metric">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="type" type="xsd:string"/>
      <xsd:element name="parameters" type="sla:Parameter" maxOccurs="unbounded"
minOccurs="0"/>
      <xsd:element name="eventHandler" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Parameter">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="value" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ServiceLevelObjective">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="type" type="xsd:string"/>
```

```
      <xsd:element name="parameters" type="sla:Parameter" maxOccurs="unbounded"
minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
    <xsd:complexType name="ActionHandler">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="handler" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ServiceLevelAgreement">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
      <xsd:element name="serviceLevelObjective" type="xsd:anyURI"
ecore:reference="sla:ServiceLevelObjective"/>
      <xsd:element name="action" type="xsd:anyURI"
ecore:reference="sla:Action"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```