SALmon - A Service Modeling Language and Monitoring Engine

Viktor Leijon* and Stefan Wallin
Luleå University of Technology Skellefteå
SE-931 87 Skellefteå Sweden
Email: leijon@ltu.se, stewal@ltu.se

Johan Ehnmark
Data Ductus Nord AB
SE-931 31 Skellefteå Sweden
Email: johan.ehnmark@dataductus.se

Abstract

To be able to monitor complex services and examine their properties we need a modeling language that can express them in an efficient manner. As telecom operators deploy and sell increasingly complex services the need to monitor these services increases.

We propose a novel domain specific language called SALmon, which allows for efficient representation of service models, together with a computational engine for evaluation of service models. This working prototype allows us to perform experiments with full scale service models, and proves to be a good trade-off between simplicity and expressive power.

1 Introduction

Operators want to manage services rather than the network resources which are used to deliver the services. This change of focus is driven by several factors; increased competition, more complex service offerings, distribution of services, and a market for Service Level Agreements [14].

A result of this transition is an increasing need to predict, monitor and manage the quality of the service that is delivered to the end users. However, the complexity of understanding and modeling services is a serious obstacle.

We want to find a way to model Services, Service Level Agreements and the structure underlying them.

Service modeling is intrinsically hard, since we need to express calculations, types and dependencies. Current UML-based approaches tend to hide this without really providing the expressive strength needed. On the other hand, using traditional object-oriented programming languages gives the expressive strength but creates a gap between the model and the domain experts. Time-dependent calculations are often complicated or unnatural to express in these languages.

*This work is partly supported by the EU's Sixth Framework Programme project SOCRADES.

An implementation challenge is to manage the *volume* of service types and instances. Service providers have large infrastructure and service portfolios. There can be several million cells, edge devices, areas and customers.

Managing a large number of object instances with calculated Key Performance Indicators, including indicators which are calculated over time intervals, have computational challenges which are not addressed in current solutions. Time is an inherent dimension in service monitoring and SLA management for several reasons. We need to be able to manage late arrival of data, there may be delays between the collection of a key performance indicator and its introduction into the SLA system for instance due to batching. The actual time-stamp must be used in the overall calculation of status which may require recalculation. Operators also want to make "time-journeys", looking backwards and forwards to understand how service quality has developed. Furthermore, SLAs contain time variables in the form of requirements on time-to-repair and availability measurements.

It is vital to be able to provide different views for different users. Naive attempts to model services use a tree structure where Key Performance Indicators are aggregated upwards in the tree. However, different roles in the organization require different types of aggregation views: per customer, per site, per area, per service, and ad-hoc grouping of service instances.

The main components of our solution are a dedicated service modeling language and a run-time environment for calculating the service status. The language is an object-oriented functional language tailored to the domain-specific requirements.

This paper makes the following main contributions towards a useful service monitoring engine:

- We give an overview of the design considerations that went into SALmon, a novel language for writing service descriptions (Section 2).
- This language has been implemented in the form of a prototype implementation of a calculation engine that

we discuss in Section 3

- Finally we examine a few typical scenarios and how they can be handled in our system (Section 4).

2 The modeling language

We employ a tailor-made programming language for defining services and service level agreements. This enables us to create services using the well understood methods of program construction. The language has two main purposes: first, it will define the structure of the model, and second, it will define the relationship between parameters and determine how they will be computed.

The language is a simple functional language for defining calculation rules. Calculations are associated with properties in object to facilitate the object-oriented structure of a service model.

Due to the nature of service modeling, the programming language must be able to treat time as part of the normal syntax: all variables are seen as arrays indexed by a time stamp. It is possible to use the time-index syntax to retrospectively change the value of variables.

List comprehension and an extensive set of built-in functions provide the power needed to express complex models. To make the language more tangible we present a simplified example taken from a model of a GSM network, see Section 2.3. The language has two fundamental layers: the Definition Layer and the Instantiation Layer.

2.1 Definition Layer

The definition layer defines the classes and calculations in the model. Core concepts that we want to represent as classes are Services and SLAs. Classes have inputs, anchors, attributes and properties:

Inputs define a time-indexed variable that is mapped to an external data source. Typical external sources are probes, alarms, performance data and trouble-tickets.

Anchors label connections to other class instances and hence provide the basis for building structures from service objects.

The definition layer only defines the name of the anchor and its multiplicity, so that an anchor is defined to have either exactly one anchored instance *or* zero or more.

Properties are values that can be left undefined in a class definition to yield an abstract base class. Properties can be defined or redefined in sub classes to model differentiated service levels.

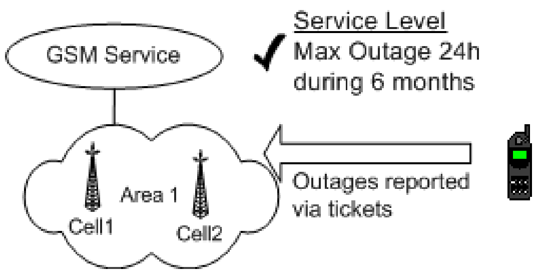

Figure 1. Sample Model

Attributes define calculation rules for parameters in a strict purely functional language.

The calculation rules have knowledge of which instance of the attribute is being evaluated, and can use that together with attributes, properties and other anchored objects to calculate values.

Code reuse is facilitated through an inheritance system where subclasses can override and redefine attributes and properties

The definition layer cannot create new objects, only define classes. The sources for inputs are not defined here. Different systems can feed the same input and using undefined inputs will result in undefined results.

2.2 Instantiation Layer

The instantiation layer creates instances of the service classes, assigns properties and establishes connections between instances through anchors.

The anchoring of instances creates a directed graph: $G = (V, E)$ where the vertices V are service objects and edges E are connections to anchors. The graph may be connected or disconnected. Common special case for service models are trees and forests.

SALmon has dedicated constructs to ease instantiation and anchoring of instances. Since we are working with a large amount of service instances and relationships, there are dedicated iterator constructs to simplify the process.

2.3 Example

We illustrate our language with a simple model with service objects for a mobile network. The purpose of the model is to provide SLAs for mobile voice services in dedicated areas. The input data source is trouble-tickets which cover both technical problems derived from alarms and customer complaints. A sketch of the sample models is in Figure 1.

Four classes are defined in Figure 2. The first class, `GSMCell`, defines a single input: the number of open trou-

```
class GSMCell
  input openTickets
  ok = (openTickets == 0)

end

class GSMArea
  anchor* cells
  ok = allTrue cells.ok
end

class GSMService
  anchor* areas
  ok = allTrue areas.ok
end

class GSMServiceLevel
  anchor service

  property OutageMeasurementPeriod
  property MaxOutageTime

  okSLA = downtime < MaxOutageTime
  downtime =
    totalFalseTime service.ok@(NOW,
         NOW-OutageMeasurementPeriod)
end
```

Figure 2. Classes for Areas and Cells

ble tickets associated with the cell. The boolean attribute `ok` is true only when the number of open tickets is zero.

The second class, `GSMArea` defines an anchor point for cells. It also defines another attribute `ok` which depends on the `ok` attribute of all anchored cells. When the area is instantiated it is anchored to the cells that cover an important area for an SLA customer, such as an enterprise main office.

The third class, `GSMService` aggregates areas into a general service perspective.

The fourth class `ServiceLevel` contains rules to calculate downtime and conformance to a service level agreement defined by properties. This represents the service sold to the end-customer, and downtime is calculated as the total time with open tickets.

We define two different service levels in Figure 1, `GSMServiceLevel1` and `GSMServiceLevel2`, which are subclasses to `GSMServiceLevel` where the properties have been fixed.

We are now ready to show the instantiation of a small service in Figure 4. It builds a service level `service1` which monitors a `GSMArea` called `area1` made up of two

```
def GSMServiceLevel1 =
GSMServiceLevel(MaxOutageTime => 24h,
 OutageMeasurementPeriod => 6 months)

def GSMServiceLevel2 =
GSMServiceLevel(MaxOutageTime => 72h,
  OutageMeasurementPeriod => 6 months)
```

Figure 3. Service Levels for GSM Service

```
create GSMCell cell1
create GSMCell cell2
create GSMArea area1
create GSMServiceLevel1 service1

connect area1.cells cell1
connect area1.cells cell2
connect service1.areas area1
```

Figure 4. Model Instantiation

cells, `cell1` and `cell2`. This example corresponds to a customer who has bought a service level agreement for their main office with a maximum outage of 24 hours per six month period.

Service monitoring needs to be integrated with external tools such as alarm and trouble ticket systems to notify operators about problems. This is handled by allowing external systems to subscribe to attributes. This mechanism also allows us to separate the presentation in the user interface from the design of the calculation engine.

3 Prototype implementation

We have implemented an early prototype of the SALmon language runtime and interpreter using the Java^{TM} J2SE Framework [10] and the ANTLR parser generator [8].

3.1 Classes

In the current implementation service models can be built from the basic building blocks:

Classes with inputs, anchors, properties and attributes.

Class inheritance where base class attributes can be overridden.

Property values can be fixed through a declaration similar to class inheritance.

```
class Service
  anchor system

  // Pass on the status attribute of the
  // instance anchored to system at the
  // time of the evaluation.
  currentStatus = system.status@NOW

  // Request the status of the last day
  // and return the worst one.
  dailyStatus = worstOf
           system.status@(NOW, NOW-1day)
end
```

Figure 5. Time variable evaluation in attribute expressions.

3.2 Expressions

Inputs and attributes can both be seen as *lists* of time-stamped values. In this sub-section we will refer to inputs and attributes as *time variables* viewed as lists of tuples (V, T) where V is the value and T is the time-stamp. With this view we abstract the fact that the values of inputs are available as semi-static data from external sources while attributes are calculated on demand by the runtime engine.

The expression for an attribute evaluates using the other available time variables, namely

- Inputs of the same class.
- Attributes of the same class.
- Attributes of other instances connected through an anchor.

The evaluation is performed by time-indexing. The current implementation restricts time indexes to constants or constant functions of the `NOW` parameter. Intervals of a time-variable can also be retrieved by specifying a time range. Examples of how time variables are evaluated are given in Figure 5

The need to handle lists of values arises as a consequence of two things: anchors aggregating multiple sub-service instances *and* processing time-intervals of inputs or attributes.

The list comprehension is provided through the common higher order list processing functions `map`, `fold` and `filter`:

map applies a unary function on all items in a list and returns a list of the result.

fold reduces a list into a single value by recursively applying a binary function on a list, for example when summing a list of numbers.

filter takes a list and returns only the values accepted by a predicate or unary boolean function.

The function arguments of higher order functions can be supplied either as an anonymous function or a named helper function. Helper functions depend only on their explicit arguments, and as such can be considered as purely functional. All functions are call by value. Basic operations for arithmetic, boolean logic and comparison are also implemented.

3.3 Execution engine

The runtime implementation provides basic functionality to load definition layer source files, create instances of classes, associate inputs with external data sources, connect instances through anchors, and request values of attributes of created instances.

All computations of the runtime begin with the request of an attribute value. The expression of the attribute is internally computed within a stack-based machine which computes a function after evaluating its arguments. This makes the attribute the fundamental runtime calculation unit.

The default state of the runtime is a resting state. As a request for an attribute at a certain time-stamp may depend on the calculation of other attributes, this will result in what can be considered a directed graph of calculation units where non-connected units can be calculated independently and hence in parallel.

In the prototype all data mapped to inputs reside in a database. Attaining satisfactory database performance is one of the main issues under investigation.

4 Scenarios

This section illustrates how to apply SALmon for fulfilling a few typical requirements on Service Monitoring systems.

4.1 Service Levels

Service Levels are modeled as normal classes which conform to “best practice” from ITIL and TM Forum standards and have a standard set of attributes (e.g. Figure 2).

They are associated with the operational objects through an anchor. It will often be desirable to have a high level SLA which aggregates the various SLAs into a single unit. This is easily expressed in SALmon since we use classes to model SLAs and Service Levels.

```
// Checks if outage is longer than window.
outageOk = binThreshold
     ((timeFilter NOW) - outageDuration)

// The percentage of remaining time.
timeRemain =
 ((timeFilter NOW) - outageDuration) /
 (timeFilter NOW)

// Helper function which returns
// the correct service window.
timeFilter t =
      if hour(t)<4 or hour(t)>19 then
        4h
      else
        2h
```

Figure 6. Service Hours Calculation

A common feature of SLAs is that they are measured on a periodic basis. The example below illustrates how outage during the current month can be calculated by summing the parameter h over the last month:

```
outage = sum h NOW month(NOW) 1m
```

4.2 Outage and Service Hours

Calculations need to be affected by time windows. For example, it might be okay to take a piece of equipment out of service if the customer is informed, or an SLA might only apply during office hours.

Figure 6 shows how to use a time filter to calculate outage. The time filter defines a longer service window in between 7 p.m. and 4 a.m.

4.3 What-if scenarios and Goal-Seeking

One might want to see how a certain action or change of parameter value changes the overall Service Quality. The result should show the affected service components and the resulting calculated values.

To solve this requirement, we have the ability to take a snapshot of the state of the system, making a copy-on-write version. Simulation input can then be applied to the copy in order to study the effects.

A straight-forward example is to simulate a big network outage. The simulation input is then driven by alarms and/or work orders in the trouble-ticket system that will affect inputs in the service models.

The opposite of "what-if" is *goal-seeking*. If an operator wants to increase the measured key performance indicators of a service, how do we find the needed changes in the low level inputs? The solution to this is to translate the model into an expression that can be evaluated in a logical framework which provides goal-seeking mechanisms. Important in this application domain is that we only need good-enough proposals, not the exhaustive list or necessarily the optimal one. The goal-seeking can be done interactively with human intervention until a good-enough change is found.

4.4 Different views

It is important to be able to provide different views for different roles such as customer care, technical maintenance, and marketing.

The "core" model is a tree and the other views are variants of the tree structure like grouping cells in another way then areas and offices, for example by type and revision.

4.4.1 Modeling of service tools

The example below shows a model which mimics the OpenView Operations Service Navigator tool [5]. It is a simple, yet useful tool to model services in a tree structure. Every node has an alarm status. Furthermore, propagation rules state how severities should propagate to parents and calculation rules specify how a parent node should calculate its alarm state based on its children.

```
class SNNode
   anchor* children
   property propRule, calcRule
   property name
   ownStatus = OK
   status =
     snFunc propRule ownStatus children
```

5 Related work

One of the most important sources for service and SLA modeling is the SLA handbook from TM Forum [11]. It provides valuable insights into the problem domain but not to the actual modeling itself.

TM Forum has also defined an accompanying service model, SID, "System Information Model" [12]. SID is comparatively high level and models entities in telecom operators' processes. However, SID is being refined and moving closer to the resources by incorporating CIM [3].

The Common Information Model, CIM, has an extensive and feature-rich model including a modeling language *MOF* (Managed Object Format). Key strengths in CIM are the modeling guidelines and patterns. However, CIM faces some major challenges since the UML/XML approach tends to create unwieldy models. It is also aimed more at instrumentation than end-to-end service modeling.

Some of the major players behind CIM are now working on the "Service Modeling Language", SML [13]. SML is used to model services and systems, including their structure, constraints, policies, and best practices. Each model in SML consists of two subsets of documents; model definition documents and model instance documents. Constraints are expressed in two ways, XML schemas defines constraints on the structure and contents whereas Schematron and XPath are used to define assertions on the contents.

An interesting feature is that SML addresses the problem of service instantiation by providing XSLT discovery transforms to produce instances from different sources. Other attempts exist to specify service models as component interaction with UML collaborations [9]. This kind of service modeling serves purposes closer to the design of systems than service models for QoS metrics.

A simple and pragmatic model for a general service model is given by Garschhammer et al. [4]. This work serves as a guide for modeling and identifies several important research areas.

SLAng [6] is a language focused on defining formal SLAs in the context of server applications such as web services. It uses an XML formalism for the SLAs. SLAng identifies fundamental requirements needed in order to capture SLAs but differs from our current effort in that it "focuses primarily on SLAs, not service models in general".

When it comes to programming languages with an inherent notion of time, Benveniste et al. [2] give an overview of the synchronous languages. These languages have the concepts of variable relationships and of computing values based on the previous value of a variable. However, they have no notion of retaining values after the computation, and they have a discretized notion of time.

Perhaps most closely related to SALmon is the notion of stream data managers [1], which take a more database oriented approach to the problem. This makes their syntax less suitable for service models, and means that they have a stricter view on time progress. However, a lot of the underlying work may be reused in the current setting.

6 Conclusion and Future Work

We have demonstrated a language for writing service models, and shown the design of a prototype calculation engine. Further, we conclude that the proposed system is a good match against real-world scenarios.

In the future, the scaling and caching that is made possible by the parallel structure of the language should be further examined. This work has been started [7], but deserves more attention.

The database layer should be updated to handle large numbers of concurrent but simple requests for input data at specific times or intervals. Since the requested data might be for a single primitive data type the overhead for each request is critical.

Finally, the user information visualization should be examined. With the ultimate goal of being able to manage large service structures, it is desirable to present relevant information in an efficient manner.

References

[1] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. Technical report, Stanford, 2004.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[3] DMTF. CIM Specification v2.15.0. Technical report, Distributed Management Task Force, 2007.

[4] M. Garschhammer, R. Hauck, H. Hegering, B. Kempter, I. Radisic, H. Rolle, H. Schmidt, M. Langer, and M. Nerb. Towards generic service management concepts a service model based approach. *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 719–732, 2001.

[5] HP. *HP OpenView Service Navigator*. URL: http://h20229.www2.hp.com/products/servnav/index.html, 2008.

[6] D. Lamanna, J. Skene, and W. Emmerich. SLAng: A Language for Defining Service Level Agreements. *Proc. of the 9th IEEE Workshop on Future Trends in Distributed Computing Systems-FTDCS*, pages 100–106, 2003.

[7] V. Leijon, P. A. Jonsson, and S. Wallin. A declarative service modeling language with efficient caching. In *Practical Aspects of Declarative Languages (PADL'09)*, 2009. **submitted**.

[8] T. Parr. ANTLR parser generator. Accessed 14th of July, 2008. `http://www.antlr.org/`.

[9] R. Sanders, H. Castejon, F. Kraemer, and R. Bræk. Using UML 2.0 collaborations for compositional service specification. *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2005.

[10] SUN Microsystems. J2SE 5.0. Accessed 14th of July, 2008. `http://java.sun.com/j2se/1.5.0/`.

[11] TM Forum. SLA management handbook. Technical report, TM Forum, 2004.

[12] TM Forum. Shared information data model. Technical report, TM Forum, 2005.

[13] W3C. Service modeling language. http://www.w3.org/TR/sml/, May 2008.

[14] S. Wallin and V. Leijon. Multi-Purpose Models for QoS Monitoring. In *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, pages 900–905. IEEE Computer Society, 2007.