FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Using Domain-Specific Languages for Event-Based QoS Monitoring in Service-Oriented Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Matthias Irlacher
Matrikelnummer 0225106

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:	Univ.-Prof. Dr. Schahram Dustdar
Mitwirkung:	Dipl.-Ing. Dr.techn. Ernst Oberortner
	Mag.rer.soc.oec. Philipp Leitner

Wien, 15.11.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Matthias Irlacher
Rustengasse 8/12, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

As the paradigm of Services-Oriented Architecture (SOA) gets more widespread, the quality of the used services also gets more important. Not only is this Quality of Service (QoS) important to human users, but to other services. Monitoring and automatic checks for violations of Services Level Agreements (SLAs) are a necessary and interesting topic.

This thesis demonstrates and evaluates a system for the measuring of service quality using a centralized monitor and a minimal-invasive framework at the clients and services. It uses three domain-specific languages (DSLs) to describe SLAs (SLADSL), define measuring points and their context (MPDSL) and create metrics based on the defined measuring points (MDDSL). In addition to the languages, a system for distribution the measuring point information is proposed. The concrete measurement takes place at the clients and services and the measured raw data is sent to the monitor. The monitor uses transformations, generated from MDDSL statements, to calculate metrics from the raw events in a Complex Event Processing (CEP) system.

The goal of this thesis was to find a fast, event-based approach to define and measure QoS properties, based on an existing approach by Ernst Oberortner. It had to be expressive, extensible and at the same time try to keep the influence and dependence on the measured systems as small as possible. Amongst other things, this should be achieved by using a separation of SLA, metric descriptions and measuring point definitions and the possibility to change SLAs and metric descriptions at runtime. The evaluation of the approach shows a minimal influence at the measurement locations when using interceptor-based measurement. The central monitor proves to be a bottleneck, but a possible solution is proposed by splitting the monitor into multiple systems. The concept of separate MPDSL and MDDSL allows to define a variety of QoS properties and enables great extensibility, even including other measurement systems. An evaluation tests the prototype and discusses some further improvements of the system.

Kurzfassung

Da das Paradigma der Services-Oriented Architecture (SOA) sich immer weiter verbreitet, wird auch die Qualität der benutzen Services immer wichtiger. Diese Quality of Service (QoS) ist dabei nicht nur wichtig für menschliche Nutzer, sondern auch für andere Services. Das Überwachen und automatische Überprüfen auf Verletzungen von Services Level Agreements (SLAs) sind ein notwendiges und interessantes Thema.

Diese Diplomarbeit demonstriert und evaluiert ein System, welches Servicequalität mittels eines zentralen Monitors und eines minimal-invasiven Frameworks bei Clients und Services misst. Sie verwendet drei domain-specific languages (DSLs) um SLAs zu beschreiben (SLADSL), Messpunkte und ihren Kontext zu definieren (MPDSL) und Metriken, basierend auf den definierten Messpunkten, zu erstellen (MDDSL). Zusätzlich wird ein System zum Verteilen der Messpunkt-Informationen vorgeschlagen. Das konkrete Messen findet bei Clients und Services statt, die gemessenen Rohdaten werden daraufhin an den Monitor gesendet. Der Monitor benutzt Transformationen, die aus MDDSL Beschreibungen generiert wurden, um in einem Complex Event Processing (CEP)-System Metriken aus den Rohdaten zu generieren.

Das Ziel dieser Diplomarbeit war es, einen schnellen, event-basierten Ansatz zu finden, um QoS Properties zu definieren und zu messen, basierend auf einem Ansatz von Ernst Oberortner. Er musste ausdrucksstark und erweiterbar sein und gleichzeitig versuchen den Einfluss auf und die Abhängigkeit von dem gemessenen System so klein wie möglich zu halten. Unter anderem sollte dies durch eine Trennung von SLA, Metrikbeschreibung und Messpunktdefinitionen, sowie durch die Möglichkeit SLAs und Metrikbeschreibungen zur Laufzeit zu ändern, erreicht werden. Die Evaluierung dieses Ansatzes zeigt einen minimalen Einfluss am Messort bei Benutzung einer Interceptor-basierenden Messung. Der zentrale Monitor erweist sich als Flaschenhals, doch eine mögliche Lösung, durch Auftrennen des Monitors in mehrere Systeme, wird vorgeschlagen. Das Konzept der getrennten MPDSL und MDDSL erlaubt uns eine Vielfalt von QoS Eigenschaften zu definieren und ermöglicht gute Erweiterbarkeit, sogar durch das Miteinbeziehen anderer Messsysteme. Eine Evaluierung testet den Prototypen und diskutiert weitere Verbesserungen des Systems.

Contents

1. Introduction

As more and more companies discover the advantages of the service-oriented paradigm, the term Services-Oriented Architecture (SOA) recently became very widespread. SOA describes an architectural approach, which uses independent services to build a software system. Currently Web services are a very popular form of service. As SOAs get more popular, the need for contracts, specifying different kinds of requirements on a service, gets more and more attention. Services Level Agreement (SLA) are such contracts which allow clients to determine a minimal provided service quality. This quality is called Quality of Service (QoS) and describes non-functional characteristics of the service. Some QoS attributes are declared by the provider, some can be observed and some can even be declared by the service client [118]. They can be discriminated by the place at which they are measured or based on certain predefined classes (such as security, availability or reliability) as used in many taxonomies. Another possible differentiation is between general QoS attributes and those specific to a certain domain or even application. Taxonomies show that there is a great amount of different general metrics (e.g., [68], [110]), but there is a possible infinite number of application-specific QoS properties. Measuring, collecting and processing such a wide range of heterogeneous QoS attributes in a single system is a interesting and difficult task. The measured QoS can not only be used for SLA violation detection, but for many other purposes (e.g., selection of services, self surveilance or testing). As such, monitoring of service quality is an important topic for both service providers and clients.

1.1 Motivation

Measuring QoS is an important and complex task in any reasonably big distributed system based on the SOA paradigm. A useful system for monitoring of non-functional properties of Web services should allow to monitor a broad spectrum of parameters and be extensible concerning the specification of new measurement methods.

[118] mentions three types of QoS metrics, based on their origin:

- *Provider-advertised metrics* are specified by the service provider (e.g., security).
- *Client-advertised metrics* are based on clients evaluation and feedback (e.g., reputation).
- *Observable metrics* can be computed by monitoring the service or client in an objective way (e.g., processing time, latency).

This thesis concentrates on observable metrics, allowing a detailed specification, especially of performance-based metrics. In fact, [118] argues that the majority of metrics are observable. The flexibility of the framework also enables input of custom provider-advertised or client-advertised metrics.

Non-functional properties can be determined at different times in the lifecycle of a service. Some properties are defined at design time, for example by choosing a specific authentication mechanism. Others can be determined when the system is running, through active polling or passive monitoring. The running system can be monitored before publishing it, to test its behavior or a production system can be monitored while in use to determine its characteristics under load. Monitoring a production system can be used for a variety of challenges. The QoS information of one's own services can be used to start adaptive actions to ensure future QoS or for SLAs negotiation. QoS information of foreign services can be used for service selection, reputation mechanisms or determination of penalties in case of contract violations. As every monitoring system adds an overhead to the system (See the "invasion problem" in [115]), it is very important to evaluate the solution, also determining its runtime behavior under load. Contracts can change and, especially if monitoring a production system, a restart is not always a feasible option. Therefore the solution in this thesis supports the runtime changes of SLAs as well as of measuring descriptions.

While [42] mentions the unlimited number of possible metrics, it refers particularly to QoS metrics specific to an application domain. But even "standard" metrics, such as *availability*, can be expressed in different ways. From experience, [59] notes that "non-modifiable textbook definitions" of metrics would not be accepted by providers and customers alike. Therefore it is of importance to specify those metrics in high detail. The possibility to explicitly define how certain QoS parameters should be measured also allows to add custom metrics easily. This helps with application-specific QoS metrics and enables the computation of complex metrics with conditional expressions, calculations, different aggregations over different intervals, etc. As a detailed description is in danger of being too dependent on a specific service architecture, another goal is to stay as independent of the architecture as possible. This current solution enables this by using the concept of parametrized Measuring Points (MPs) which can be easily exchanged in the measurement description.

This custom metrics support requires advanced computation mechanisms. While some solutions use temporal logic (see Section 3.2), the approach in this thesis uses domain-specific languages (DSLs), programming languages specifically designed for a certain domain (see Section 2.5). The use of DSLs has many advantages. It supports transformations from syntax to a model and at the same time allows for a good separation of syntax and domain model. In the form of *internal DSLs* it allows to reuse the language components of a host language (such as loops, references). The *Frag* [101] programming language (see Section 4.3) supports such embedding as well as the definition of a proper syntax in case of external DSLs. It was already used for the Quality of Service Language (QuaLa), which built a foundation for this thesis, and can be as easily embedded into Java as being used for code generation.

1.2 Problem Statement

One of the main targets of the thesis is to build a system which uses multiple DSLs for different stakeholders, which together allow to surveil the QoS of Web services. The SLA and metric specifications should be allowed to change at runtime, as a restart of a running web application is not always an option. While achieving this, it should also remain flexible and extensible and should be easily integrated into a running system. This thesis mainly discusses a monitoring approach, which is supporting these requirements.
It incorporates two research questions from [115]:

- What are useful mechanisms for the monitoring of QoS?
- How can we collect, analyse and handle the incoming monitoring events?

There is a lot of literature about validation of SLAs, including the collection of QoS attributes, but how to obtain QoS information of a concrete service is often overlooked [118]. This thesis aims to provide a solution for monitoring Web services, from a detailed specification of how to measure a QoS attribute over the validation of SLAs to the notification of necessary adaptation components or responsible authorities.

1.3 Contribution

Besides providing an overview over current QoS monitoring approaches in SOA, including some uses of monitoring in service management, this thesis presents an approach to QoS monitoring of Web services. To specify the different documents it uses three DSLs (see Section 2.5). The core system uses a Complex Event Processing (CEP) engine (see Section 2.4), a system for processing and filtering events as well as aggregating simple events to complex events.

The presented approach is based on an early version of the QuaLa [79] by Ernst Oberortner (see Section 4.4), with modifications and additions to support our goal. The QuaLa system allows to measure performance-related QoS parameters (e.g., round trip time, marshaling time or processing time) by defining the phases at which they should be measured. It uses two DSLs for monitoring QoS of Web services at the client- and service-side using a centralized monitor. The first DSL, called "High-level QuaLa" is used to describe SLAs, consisting of a list of services, conditions and actions to be taken if a condition is violated. The second, called "Low-level QuaLa", is used to describe technical aspects of the services. These include the technical details of the measured service (such as operations, address, etc.) as well as the structure of the framework that is used by the service. Based on these DSLs, code for interceptors at the client and service level is generated, using templates. For SLA verification, it uses CEP in the centralized monitor.

The solution presented in this thesis uses the QuaLa by modifying and extending it. The contribution of this thesis regarding the proposed method of monitoring can be split in four parts:

- A modification to the High-Level DSL of the QuaLa, extended using some syntactic and semantic additions. It is used for describing SLAs. To better represent the usage in our context, the modified DSL is called Services Level Agreement DSL (SLADSL) in this thesis. Instead of generating Java code, it is now interpreted at runtime and translated into CEP event transformation rules and queries for notifications and constraint verification.
- The Low-Level DSL of the QuaLa was heavily modified and reduced. It is now used to define MPs, their parameters and their technical surroundings, used for addressing a single point of interest in an application. As its purpose and model has changed heavily, the new DSL is now called Measuring Point DSL (MPDSL).
- A new Measurement Description DSL (MDDSL) for the detailed description of measurements. It allows to define QoS metrics using parametrized Measuring Points (MPs) from the MPDSL and predefined functions for calculations, time dependencies, aggregation, etc. Each metric definition is translated into multiple CEP transformation rules. The new DSL strives to be as system-independent as possible, concerning the used Web service framework, while keeping a high degree of expressiveness.
- A framework for service requestors and providers, which aims to blend naturally into the monitored service, trying to cause as little impact on the service architecture as possible. It currently supports MPs for performance-based metrics at the middleware of the client and service provider using the interceptor pattern, MPs for repeated framework-initiated events and basic support for application-specific metrics. The core of the framework architecture consists of a central monitor and a system for distributing information on MPs to different services and clients at runtime. The monitor is responsible for processing the DSLs and transforms them to CEP transformation rules and queries. It also houses the CEP engine. It collects incoming events and uses the transformations to calculate metrics and queries to check for SLA violations. If a violation is detected, its `action subsystem` notifies the corresponding authorities, declared in the SLA.

For evaluating the solution, a prototype is implemented using Java and the the Apache CXF [6] framework, as the flexible interceptor architecture allows to hook into the various phases of message processing (see Section 4.1). Esper [31] is used as a CEP solution. A Web interface is implemented for visualization of historic QoS information and for managing of services, measurement descriptions and an SLA. In an evaluation, the prototype is used to determine the performance impact of the solution and the whole approach is discussed.

This thesis does not aim to build a complete management system, but concentrates on the monitoring aspect. The main focus lies on event-based monitoring, detailed definition of metrics as well as run-time changes of metrics and SLAs, while designing the system to assist in other requirements. While it provides a solid base for adaptation, service selection and SLA negotiation, those topics are out of its scope.

1.4 Organization

The rest of the thesis is organized as follows:

- Chapter 2 describes the state of the art of the topics mentioned in this thesis. It reflects current knowledge about SOA, with a focus on technologies currently used in Web services. Afterwards, the basics of QoS are discussed, including SLAs, metrics and some measurement approaches. Some of the concepts of CEP are described in another section. The last section of the chapter deals with DSLs. It discusses their general implementation structure, generic patterns and also mentions some of the benefits and risks of using them.
- Chapter 3 shows current research efforts in the field of QoS monitoring and QoS metric definition. In the first section, different monitoring approaches are presented as well as some uses for monitoring and other related ideas. The second section describes some approaches how custom QoS metrics can be represented. Both sections include descriptions of various measurement or even management architectures for Web service.
- Chapter 4 shows the foundation this thesis is build upon. At first the frameworks used in this thesis and prototype are described: *Apache CXF* and *Esper*. The last two sections deal with the Frag language, which is used for defining (and some transformation) of the presented DSLs and the QuaLa, which some of the DSLs in this thesis are based on.
- Chapter 5 describes the design of the proposed solution. The general architecture is shown, as well as an explanation of the used DSLs.
- Chapter 6 presents the implementation details of the prototype, which was developed for this thesis. After describing the general structure and the process of DSL transformation, some of the encountered problems are presented.
- Chapter 7 first evaluates the solution by measuring the performance impact of the framework and the monitor, as well as the currently used metrics. Afterwards a quantitative evaluation discusses challenges in the proposed system and limitations of the approach.
- Chapter 8 provides a short conclusion of this thesis and also mentions some possible future work, based on the current progress.

2. State of the Art

This chapter discusses the technologies, which build the foundation of this thesis. At first, *Services-Oriented Architectures* are explained, followed by a description of *Web services* as a method of realization. A short overview of *Quality of Service* completes the part on Services-Oriented Architectures. At last, the current state of the art in *Complex Event Processing* and *domain-specific languages* is presented.

2.1 Service-Oriented Architectures

SOA is an architectural approach which uses services to build a software system. It is independent of platform- or language-specific technical frameworks [85]. In the "Reference Model for Service Oriented Architecture" [67], the Organization for the Advancement of Structured Information Standards (OASIS) describes SOA as "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains". Those *capabilities* are offered by services (see Subsection 2.1.1). [82] defines the process of SOA more directly as "a way of reorganizing software applications and infrastructure into a set of interactive services". It builds the base of Service-Oriented Computing (SOC), which is described as "the computing paradigm that utilizes services as fundamental elements for developing applications/solutions".

2.1.1 Services

Services are one of the fundamental elements of SOA. They are autonomous, platform-independent and can be described, published, discovered, and loosely coupled in new ways [84]. OASIS defines them by referring to the *capabilities* from their SOA definition:

> "A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description." [67]

To be able to leverage the full potential of an SOA, [82] states that services should

- be *technology neutral*: Services must be invokable through common standard technologies, available on almost all systems or platforms.
- be *loosely coupled*: They must not depend on knowledge of internal structures or conventions of the invoking or invoked service.
- support *location transparency*: They should have their service information in a location-independent storage, like a repository, which is easily accessible to requestors.

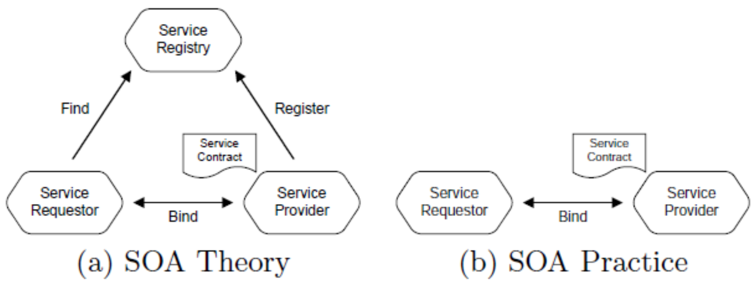


Figure 2.1: The SOA Triangle in Theory and in Practice (from [74])

When designing a software system, the question of interface granularity arises. A service operates on a business level transaction granularity rather than on a technical level [60](page XVIII). [82] even writes of services as “a business function implemented in software [. . .]”, although with a technical foundation.

While services, like components, are described by an interface, [85] notes that it is not appropriate to assume *service* is just another term for *component*, as they are independent of platform or language specific frameworks. It also distinguish *services* from *interfaces* by stating that there are extended contracts, agreements and even a particular ontological standpoint, which has an influence on a services semantics, which go beyond “a collection of function signatures” [85].

2.1.2 The Basic Service-Oriented Architecture

The basic SOA is a relationship of three roles building the so-called “SOA triangle” [74] (see Figure 2.1a):

- *Service Provider*: Service providers provide one or more services for requestors. A service can be provided by constructing an entirely new service, by composing a new service from existing services or applications(see Subsection 2.1.3) or by transforming an existing application into a service [81](page 648).
- *Service Requestor*: A service requestor is the entity, which requests a service from the provider. It does not need to know about the implementation of the service, as long as the service provides the necessary interfaces and quality properties [81](page 261).
- *Service Registry*: A service registry is responsible for storing information about services. It is also responsible for providing an infrastructure for managing of service information. This especially involves support for publishing and querying. This way the registry allows location transparency and decouples providers and requestor. As there are different ways a registry can be built, [26] provides an overview of different realizations.

Those basic components interact in three predefined ways:

- *Publish*: The service provider sends a description of the service to the registry, which stores the information into a repository for later queries.
- *Find*: If a service requestor wants to connect to a service, it can use the registry to find a matching service. Matching can be done based on a varying set of parameters, as the query languages and the stored binding data differ across registry approaches [26]. The requestor then receives the corresponding service information (including the Web service address), which allows it to bind to the specified service provider.
- *Bind*: After a service requestor acquired the binding information for the provider, it connects to the service provider. It can then use the offered capabilities of the advertised services.

In practice, registries are not commonly used [74, 83]. [74] even considers the SOA triangle broken for Web services. Without a working registry(see Figure 2.1b), location transparency is often omitted. Therefore, alternatives for some of the shortcomings of this approach have to be found (e.g., using Event-driven SOA in [83] or using *WS-MetadataExchange* [21]). An analysis of different types of registries can be found in [26].

2.1.3 Service Composition

There are two types of services: *simple* and *composite* services [82]. Simple services (or *component* services) provide concrete capabilities. Composite services combine existing services in new ways. This process is called "service composition" and is an important characteristic of SOC. A well organized architecture fosters reuse of services through combination. This combination can be done by *static* (predefined) binding or *dynamic* (run-time) binding. [74] claims that static binding can lead to inflexible and non-adaptable architectures.

Composition of services can be seen from two perspectives: orchestration and choreography.

- *Orchestration* describes the interaction of services at a message level. It allows to build an executable business process, controlled by one of the business parties [84]. In the area of Web services, the Business Process Execution Language (BPEL) [3] is an example of a language to orchestrate services.
- *Choreography* does not specify the exact business process, but describes the globally visibly message exchanges or interaction rules between multiple business-processes or end points [84]. For instance, the Web Services Choreography Description Language (WS-CDL) [58] can be used to choreograph Web services.

According to [84], the distinction is rather artificial and the consensus is that both views should be combined into a single language.

An important consideration when building a composite service is the selection of services, especially if the selection happens dynamically at run-time. The selection has a substantial influence on the quality of the resulting composite service.

2.1.4 Benefits

[29] lists some of the benefits of using an SOA, for example:

- *Leverage existing assets*: SOA can provide a layer of abstraction, which allows for continuing use of existing assets by wrapping them as a service.
- *Easier to integrate and manage complexity*: SOA moves the integration point from the implementation to the service specification. This makes integration of different systems more manageable, even across business boundaries.
- *More responsive and faster time-to-market*: The use of existing components and services shortens the software development life cycle. New business services are created and changes can be responded to faster.
- *Reduce cost and increase reuse*: Loose coupling allows for better reuse and combination. It therefore also avoids resource duplication and lowers costs.

Nevertheless, it also argues that SOA is not an universal solution and proposes a slow, incremental transformation to the new paradigm.

2.2 Web Services

The World Wide Web Consortium (W3C) describes Web services as follows:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using Hypertext Transfer Protocol (HTTP) with an Extensible Markup Language (XML) serialization in conjunction with other Web-related standards." [69]

[83] notes that Web services are the most popular type of service. According to [85], there is a bias in SOA literature towards Web architectures. Among others, [81](pages 12-19) mentions the following interesting properties of Web services: Web services can be *simple* or *complex*. Complex services are constructed by combining other services. They can be modeled in BPEL (see Subsection 2.1.3). Web services can have *functional* or *non-functional* properties. Functional properties describe the technical details of the service, such as encodings, invocation information, location, etc. Non-functional properties describe more abstract concerns as quality,

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <callID xmlns="http://compas.infosys.tuwien.ac.at/">ID_1</callID>
  </soap:Header>
  <soap:Body>
    <ns2:methodA xmlns:ns2="http://compas.infosys.tuwien.ac.at/ServiceA">
      <paramA>valueA</paramA>
      <paramB>valueB</paramB>
    </ns2:methodA>
  </soap:Body>
</soap:Envelope>
```

Listing 2.1: Example SOAP Request

performance or security. Finally, Web services can also be *stateless* or *stateful*. A stateless service carries no information about previous calls. In contrast, a stateful service keeps its state across multiple service calls. Processing of a new call is based on the state, which resulted from previous service calls.

2.2.1 SOAP 1.1

The Simple Object Access Protocol (SOAP) is a standard of the W3C. Version 1.1 is specified in [13]. Web services interact via message passing through the use of *SOAP messages*. SOAP messages are XML-based, complex data types are described in XML schema. They are therefore platform and technology neutral. A SOAP message can be sent over various communication protocols. This is implemented through the use of *bindings*. A binding is a way to map a SOAP message to a message of a specific transport protocol. The SOAP standard defines a HTML binding [13] where SOAP messages are sent through POST requests. Other transmission methods, such as the Java Message Service (JMS), a message-oriented middleware, are also supported via different bindings [1].

Message Structure

SOAP messages are based on XML. The root-element is named “Envelope”. It contains an optional *Header* and a *Body* element. A simple SOAP message is shown in Listing 2.1.

The *Header* serves as a means of extensibility. It can contain multiple header entries. New functionality can be added to a message by adding new entries to the header (e.g., for authentication, transaction management or payment) [13].
Each entry has two optional attributes.

- The `actor` attribute allows to define an addressee for this header entry. A message can travel over many SOAP *intermediaries* on its path to the final *recipient*. Through the actor attribute, any of those intermediaries can be target of a header entry. If an intermediary receives a message, it MUST process and remove all header entries targeted at it. If the actor attribute is omitted, the header entry is targeted at the final recipient.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <callID xmlns="http://compas.infosys.tuwien.ac.at/">ID_1</callID>
  </soap:Header>
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>exceptionText1</faultstring>
      <detail>
        <ns1:ServiceException
            xmlns:ns1="http://compas.infosys.tuwien.ac.at/Service">
          <exceptionAttributeA
              xmlns:ns2="http://compas.infosys.tuwien.ac.at/Service">
            exceptionText2
          </exceptionAttributeA>
        </ns1:ServiceException>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```
Listing 2.2: Example SOAP Fault Response

- The `mustUnderstand` attribute defines if the target actor (or final recipient) has to process the semantics of the message. If the attribute is set to “1” and the actor can not process the header, it must return a fault message. If the attribute is set to “0”, a intermediary can silently drop the header entry if it does not fully understand it.

The *Body* is the main part of the message. It can have many body entries. A body entry can be, for example, a request message, a response or a fault. SOAP defines a special remote procedure call (RPC) encoding for invocations of remote procedures. It defines how the function signature of a remote function can be mapped to a corresponding XML structure and how the parameters and return values can be encoded into XML. In case of faults, there is exactly one `Fault` element in the body. It contains a fault code, a human readable fault description and the name of the actor, who caused the fault. An optional `detail` field contains application specific error information concerning the body. If an error occurred while processing the header, it must be absent. This allows a client to check if the body was processed by looking at the existence of the detail field. Listing 2.2 shows an example of a SOAP fault.

2.2.2 WSDL 1.1

Web Services Description Language (WSDL) [17] is an XML-based standard of the W3C. It allows to describe operations, the location and the required message encoding of a service. Like SOAP, it uses XML schema to describe used data types which makes it platform and technology neutral.

Structure

WSDL has an *abstract* and a *concrete* part. This separation provides a good extensibility to different bindings.

- The abstract part defines the basic operations and messages. A `port type` is a set of operations, which in turn consist of *in*- and/or *out-messages* and *faults*. Different operation types (request-response, one-way, notification and solicit-response) can be implemented through different order or combinations. A message consists of one or more *parts*. The parts can, for example, represent function parameters in a remote call (see SOAP-Binding RPC style in Section 2.2.2)
- The concrete part describes the mapping of the abstract operations to a concrete location and protocol. The mapping to a protocol is provided by different bindings (see Section 2.2.2). Also, the used bindings are described and how the binding should map operations or types of a specific port type to the protocol (e.g., if a message should be send through multiple HTTP requests or a single HTTP request and response). Each *service* consists of *ports* which reference bindings and add additional information (e.g., an address).

Bindings

The WSDL 1.1 standard defines support for a variety of *bindings* [17]. A binding provides the mapping of abstract types and operations to concrete protocol-specific messages. One of those bindings is the SOAP (1.1) binding. The user can select, among others, which transport protocol to use (see Subsection 2.2.1), headers, body or fault for the SOAP message, a port address and a *style* for the operation. The operation style allows to choose if the message should be encoded according to the SOAP RPC encoding (see Section 2.2.1). Other bindings defined by the standard are the *HTTP binding* which supports HTTP GET and POST and a special Multipurpose Internet Mail Extensions (MIME) binding for a variety of MIME types (e.g., "multipart/related" or "text/xml")

2.2.3 Discovery

There are different methods for service discovery. [69] lists three of them:

- via *Registry*: A service can register itself at a registry, but the owner of the registry has the possibility to control who is allowed to add/modify registry entries and which information is stored. The Universal Description, Discovery and Integration (UDDI) [19] is a typical implementation of a Web service registry.
- via *Index*: In contrast to a registry, when using an index, the index owner selects which services to add. It is not authoritative, but contains pointers to authoritative information. Therefore if any information is outdated, it can be detected easily by referring to the authoritative source. UDDI can also be used as an index instead of a repository.

- via *Peer-to-Peer*: In a peer-to-peer solution, there is no centralized information store. The peers of the network communicate with each other to find a response to a query. This provides a more reliable approach, but the communication overhead can also lead to a decrease in efficiency.

2.3 Quality of Service

Literature on QoS typically speaks of two variants: QoS on the *network level*, such as telephony or media streaming (e.g., [15, 54, 88]) and QoS on the *application level*, such as Web service (e.g. [2, 72]). On the network level, typical QoS attributes include throughput or packet loss. A system which provides QoS can also be seen as a system which provides mechanisms for ensuring those attributes. On the application level, QoS describes non-functional properties of the services itself, such as cost or security specifications. While functional properties describe *what* the system does (e.g., through interfaces), non-functional properties describe *how well* it is done. They add additional information about security, response times, reliability and other qualities of the system. In SOA, QoS often describes non-functional properties of the services themselves, of the network level or mixed properties (e.g., *Accuracy*, *Latency* and *Round trip time* in [72]). Some researches see QoS as a part of non-functional attributes of a system, others treat them as the same thing. In general, there does not seem to be a consistent opinion on the exact bounds of the term.

2.3.1 Metrics

There are lots of different attributes, which can be measured in different ways. Literature provides a wide range of QoS properties. [81](page 36), for example, lists the following: availability, accessibility, conformance to standards, integrity, performance, reliability, scalability, security and transactionality.
An interesting categorization is provided by [9]. It separates *internal* and *external* attributes. Internal attributes can be calculated just by looking at the service. This includes attributes such as processing time, wrapping time or accuracy. In order to get external attributes one must look at the environment of the service. Those attributes, which [9] notes as being more important to service requestors, include response time, security or availability.
[118] splits QoS metrics into three categories: *observable*, *provider-advertised* and *consumer-rated* metrics. *Application-specific* metrics, which cross-cut the metrics from [118], are of additional interest, but are not always counted as true QoS metrics throughout literature.

- *Observable* metrics can be calculated based on objective observations from requestors or services. [118] further distinguishes between metrics on *IT level* or *business level.* On IT level, they include, for example, processing time or reliability. Metrics on business-level are often domain-specific (e.g., the accuracy of a prediction).

- *Provider-advertised* metrics are published by the provider of the service (e.g., the price for service usage).

- *Consumer-rated* metrics are calculated through evaluation of the service by the client (e.g., popularity or reputation). They provide a subjective view on the service.
- *Application-specific* or *domain-specific* metrics are used for a specific domain or application. [42] mentions, that there is an unlimited possibility of metrics. For example, [9] lists, among other QoS attributes, *accuracy* and *precision*. Those two measurements describe the deviation from the real value (accuracy) or the number of digits after the decimal point (precision) of a result.

2.3.2 Service Level Agreements

[55] defines SLAs as follows:

> "A service level agreement is an agreement regarding the guarantees of a web service. It defines mutual understandings and expectations of a service between the service provider and service consumers. The service guarantees are about what transactions need to be executed and how well they should be executed." [55]

Important parts of the SLA include, among others, Services Level Objectives (SLOs) [55,66,93], which are conditions which must be fulfilled, and penalties if they are not. If an SLO is violated, penalties depend on the use case. When selecting a service based on QoS, another service can be chosen. When using a reputation mechanism, reputation can be decreased. The penalties can also be predefined costs or other compensations.

There are two types of SLAs, when distinguishing based on the types of constraints: *Hard Contracts* set hard limits to the values allowed for a property. *Soft Contracts* come in different variations, which allow to soften those limits. As examples of soft contracts, [90] proposes the use of soft *probabilistic* SLAs by using probability distributions. [93] monitors the values of the QoS properties over time to calculate a "degree of SLA fulfillment".

2.3.3 Measurement

There are many reasons why measuring QoS can be important. It can be used to simply test performance of a service. When using compositions to measured QoS can be used to adapt to changes in service quality on-the-fly and to ensure the certain overall QoS of the composed service. If an SLA is used, clients can use the results to detect contract violations and enforce penalties. Service providers, on the other hand, can use the measured data to start countermeasures before an SLO is violated.

[117] distinguishes different approaches to QoS measurement, according to the location where the measurement is taken: *consumer*, *service* and *third person*.
This work also mentions different levels, measuring can take place at:

- *Service level*: The basic approach, which measures directly at the service or consumer.
- *Communication level*: Corresponds to the measurement by a third person. Messages between consumer and service are intercepted.
- *Orchestration level*: This approach monitors orchestrated services through mechanisms provided by the orchestration engine.

While [54] does not monitor the QoS of services, but of network segments, its categorization can also be applied to services:

- *End-to-end monitoring* measures the QoS by measuring between a sender and a receiver [28]. When monitoring services, this corresponds to the service and the requestor.
- *Distributed monitoring* uses intermediate nodes between the sender and the receiver [28]. This allows to detect possible QoS degradations at the network. While this is generally an issue with streaming, it could also detect the source of an availability problem or different steps in the SOAP transmission process when using SOAP intermediaries.

Another way to distinguish monitoring methods is used by [2]:

- *Active measurement* sends packets to a monitored service to simulate user traffic or to probe the service. When simulating user traffic, this adds additional load to the service which can affect measurements and user experienced performance. When probing the service, the results may not actually be the same as the the QoS for the user.
- *Passive measurement* only monitors traffic without sending any packets. This potentially requires to monitor a lot of traffic.

2.4 Complex Event Processing

Complex Event Processing (CEP) is a relatively new concept, which incorporates some recent views, but is based on some older ideas. This section is largely based on [65].
There are two current approaches to the processing of event streams: CEP and Event Stream Processing (ESP). The definition of the two terms is subject to a lot of confusion among readers. In [64], Luckham and Schulte write:

> "The terms ESP and CEP are conceptual classifications. They can be useful in delineating philosophies of event processing and intended applications, but do not specify precisely the underlying capabilities of event processing engines." [64]

So, both are in fact mechanisms for *stream processing*, but seen from a different perspective with difference goals. A contrary approach to stream processing is *rule engines*. [98] defines the difference of stream processing to rule engines as follows: *Rule engines* contain condition/action pairs which are named *rules*. If a condition is met, an action is fired. *Stream processing* concentrates on streaming data. It uses SQL style syntaxes with special constructs for stream-oriented processing (e.g., time windows)
[98] also defines some rules for stream processing systems:

- Data must be kept moving. This is one of the main differences to a traditional Database Management System (DBMS).
- There has to be SQL-style processing on streams.
- An implementing system has to be able to cope with stream imperfections, such as missing, delayed or out-of-order events.
- Predictable outcomes must be generated by processing messages in a predictable manner.
- It must be possible to integrate stored and streaming data (e.g., for comparison to historic events, or joins to extended data from a DBMS).
- Data safety and availability should be guaranteed for most applications.
- A distributed application needs support for *scalability*. Splitting on multiple processors or machines should be possible.
- Instantaneous processing and response should be guaranteed through high performance.

For this thesis CEP is interpreted as an event processing mechanism, which uses continuous queries on event streams or partially ordered clouds of events to query or build complex (aggregated) events. It offers extended correlation mechanisms (e.g., patterns) and supports various relationships between events, such as time, causality or aggregation.

2.4.1 Events

An event is a representation of an *activity* in a system. [65](page 88) notes three important aspects of events:

- *Significance*: The significance is the activity which is represented (or *signified*) by this event.
- *Relativity*: An activity is dependent or related to other activities through aggregation, time or causality. Events have the same relationships as their correspondent activities. The collection of those relationships of a specific event is called “relativity” of this event.
- *Form*: The form of an event is its representation in the system. This includes information about the activity it represents and its context (relativity).

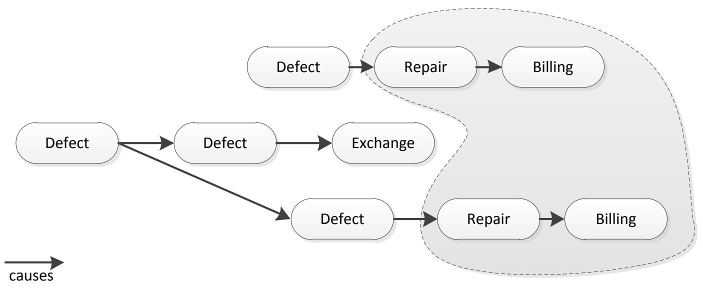


Figure 2.2: POSET of Events with Causality and a Selection (after [65]).

As already mentioned, there are three different types of relations between events [65](page 94):

- *Time*: The time relation is defined by the timestamps of the events. There may be more than one time relation between events, as there can be more clocks which are not necessarily synchronized. The time at which the events arrive at the system does not have to be the same as the time the activity occurred (it occurred *out-of-order*), this must be considered when using such a system.
- *Cause*: If an event causes another event, it has to happen before the other event for it to happen (for more detail see Section 2.4.2).
- *Aggregation*: A *complex event* is build by aggregating other events in different ways. Those so called *members* of the complex event can be simple events, generated because of the detection of an activity, or complex events.

All of those relations are transitive and asymmetric. Each builds a partial ordering, therefore the consideration of those attributes suggests a CEP solution.

2.4.2 Concepts

Some interesting concepts, related to CEP, are presented in [65]:

Event Patterns and Event Pattern Rules

An *event pattern* is used for matching partially ordered sets (POSETs) of events (see Figure 2.2). It contains of a set of events as well as causal dependencies between those events, timing, data parameters and context. It builds a template for a POSET of events [65](page 114).

An *event pattern rule* contains a *trigger* and an *action*. The trigger is an event pattern, the action an event. If the trigger matches a POSET, the action is fired and an event generated.

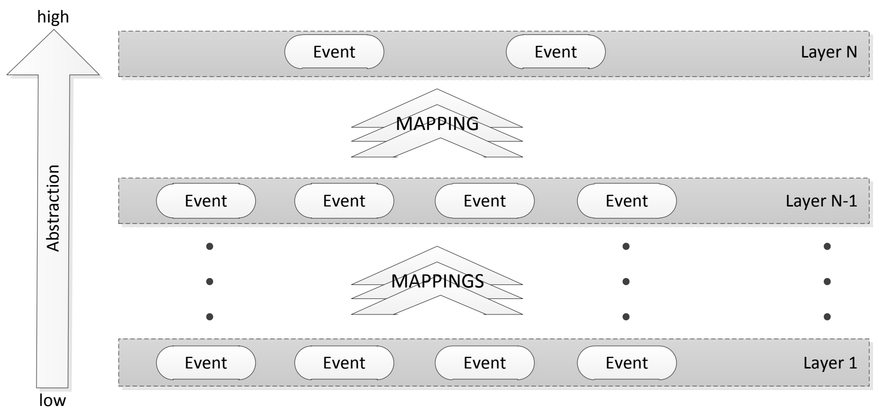


Figure 2.3: Example Abstraction Hierarchy (based on [65](page 281))

Causality

According to [65](page 241), Causality can be split into two subcategories: *static causality* and *dynamic causality*. When assuming that "activity A causes activity B", in static causality, for B to happen, A must have happened. In dynamic causality, this is not necessarily the case, without looking at the context. It is even possible that A or B happens alone.
This leads to the conclusion, that the existence of events for A and B within the system does not mean that an event with activity B happens because of A, without extra information. Therefore [65] recognizes a need for a *causality vector* for each event, which contains all its causing events.
Event pattern rules automatically define causality, at least between each trigger event and the action [65](page 120).

Event Abstraction Hierarchies

An *event abstraction hierarchy* is a hierarchical structure which describes the building of complex events in layers. Complex events can be built through aggregation of simple events or other complex events. They can be organized in hierarchical layers which represent the abstraction level of the containing events (see Figure 2.3). The higher the layer, the more abstract the meaning of events. Between those layers, mappings can be defined, to map the events of lower layers to higher layers. In an information system architecture the events on the bottom could be corresponding to technical activities, while the upper events represent business activities.

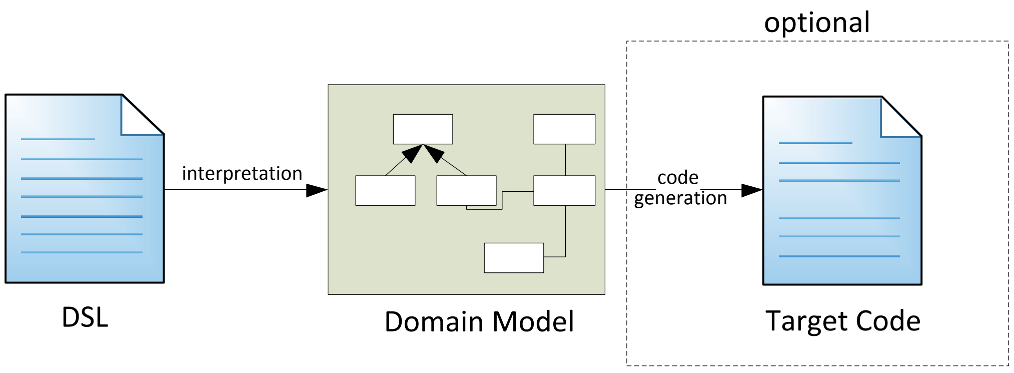


Figure 2.4: Possible Generic Architecture (after [43](page 43))

2.5 Domain-Specific Languages

A definition of domain-specific languages (DSLs) can be found in [111]:

> "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [111]

If this restriction to a problem domain is not given, it can be because some DSLs incorporate additional constructs for added flexibility. This blurs the borders between DSLs and general purpose languages (GPLs). [43](page 29) makes a good attempt at defining boundaries between DSLs and similar ideas. To differentiate between external DSLs and GPLs, configuration files or even "human jargon" is difficult. Similarly, internal DSLs can easily be confused with a normal Application Programming Interface (API). On pages 27-28 [43] notes four defining criteria for a DSL:

- *Computer programming language*: The language must be easy to read for humans, but also executable for the computer.
- *Language nature*: An important property is *fluency*. This means, that the meaning must not only come from expressions, but also from the relation between those expressions.
- *Limited expressiveness*: The language must not have too many capabilities. Additional features reduce simplicity. A DSL should have the fewest possible features to describe their *domain*.
- *Domain focus*: The focus of the language must lie on the domain, otherwise it is not useful to choose a DSL over a GPL.

2.5.1 Generic Structure

A generically structured approach is shown in Figure 2.4. A DSL is parsed, interpreted and the resulting data is fed into a domain model (called *semantic model* in [43] to separate it from the general domain model of the application). It is important to notice that this model is not necessarily the same as the DSL's syntax tree [43](page 48). Sometimes the domain model is all that is needed, but in some cases it is processed further. In *code generation*, source code for another programming language is generated from the domain model. In practice there are many possible implementation patterns.

2.5.2 Patterns

A common separation in DSLs is between *internal* and *external* DSLs:

- An *External DSL* has a discrete syntax.
  This syntax can be unique to the language, or borrowed from another language (e.g., XML). It is parsed and the semantic information is entered into the domain model. While simple approaches use a textual syntax, there are also graphical or table-like representations. The boundary is not always as clear, as [99] considers Excel [75] a DSL while [43](page 31) does not.

- An *Embedded/Internal DSL* is embedded into the code of another DSL or a GPL (the *host language*). [47] notes the basic idea of a domain-specific embedded language (DSEL). The idea behind this approach is, that an external DSL has a lot of overhead for parsing and development. If existing language infrastructure can be used, the cost of implementing a DSL can be reduced. An embedded DSL has no explicit grammar but uses the grammar of the host language for its constructs. Its syntax tree is created on the call stack, when the corresponding part of the host language is executed [43](page 50).
  The approach of [47] is appropriate for languages, which are very modular, whose functionality can be reduced by removing some components and which can be easily extended (e.g., Haskell [102], a functional programming language). There are other approaches which leave the host language unchanged but add a DSL through creative use of interfaces or other language structures. *Fluent interfaces* are such a way to implement some kind of DSL. They are created by returning a modified object from a modifying method, allowing to call yet another method and thus building a chain of method calls, sometimes similar to a natural language grammar. An example for this is the way to create named queries in the Java Persistence API (JPA) 2.0 [24].

There are many more patterns related to DSLs (e.g., [70, 97]).
[70] compiled DSL patterns and categorizes them into *design patterns* and *implementation patterns*.

Design Patterns

Design patterns can be patterns about the relation of a DSL to existing languages.

- *Language exploitation*: The DSL uses another language as the base (host language). There are three subtypes:
  - *piggyback*: Partially uses the host language.
  - *specialization*: Reduces the functionality of the host language (e.g., Haskell in [47])
  - *extension*: Extends the functionality of the host language.
- *Language invention*: The DSL is a new language.

Another base for design patterns is, how the language is described. This can be in an informal or in a formal way (e.g., through attribute grammars, rewrite rules, or abstract state machines) [70].

Implementation Patterns

Implementation patterns decide how the DSL is incorporated into the system

- *Interpreter*: An interpreter reads, decodes and executes the DSL.
- *Compiler/application generator*: The constructs, specified in a DSL, are translated into another language. There is the possibility to create only parts of the target system and let other code be written by hand. It is then very important to cleanly separate generated from hand-written code [43](page 126).
- *Preprocessor*: DSL constructs are also translated into a another language, but static analysis is only done in the target language.
  - *Macro processing*: Macro definitions are expanded to target language constructs.
  - *Source-to-source transformation*: DSL constructs are converted to target language constructs.
  - *Pipeline*: Many DSLs are built as stages in a pipeline. The output of one stage is input for another.
  - *Lexical processing*: No complicated parsing takes place, just relatively simple lexical scanning.
- *Embedding*: The DSL is embedded into a host language using only constructs of the host language (e.g., by creating data types or operators).
- *Extensible compiler/ interpreter*: A compiler or interpreter for a GPL is extended to support domain-specific optimizations or code generation.
- *Commercial Off-The-Shelf (COTS)*: Using existing tools in a restricted way for domain-specific purposes.
- *Hybrid*: Some combination of the other patterns.

2.5.3 Advantages and Disadvantages

There are many potential benefits and risks when deciding to use a DSL.
The following lists provide an overview over some of them.

Advantages

A huge advantage of DSLs is their focus on a specific domain at a certain abstraction level. DSLs should be very concise and easy to learn because of their limited expressiveness [111]. This limited expressiveness is counteracted by using appropriate formalisms for a certain domain [97]. Reusability, maintainability and testability can be improved by using a DSL [111]. The flexibility of DSLs supports a wide range of design patterns (see Section 2.5.2) for different usage scenarios. By choosing the right level abstraction, it is possible to involve domain experts more directly [111]. Domain knowledge can be reused, because it is expressed in a concrete way. This knowledge can be validated and optimized at domain level [111]. [97] also speaks of better runtime efficiency compared to "hard-coded program logic" and a modest implementation cost, although this probably depends on the implementation pattern.

Disadvantages

There are two seemingly contrary risks when using DSLs. If many DSLs are developed and used, the phenomenon of *language cacophony* [43](page 37) can occur. Too many languages have to be learned, so it is difficult to bring new people to the system. The other problem is the one of *ghetto language* [43](page 38). If a DSL is used for many different purposes, it looses its domain-specificness and develops into a GPL. The problem here is one of maintainability and of keeping it up-to-date with changes in technology, but also with finding new people for projects. Another group of problems concerns the right level of abstraction. It is difficult to find the right scope of a DSL and it is also difficult to find the right balance between DSL and GPL constructs [111]. [97] finds DSLs to not yet be well integrated into the software engineering process. Training, design, implementation and maintenance add extra costs to the process, while limited tool support makes it even more difficult [97]. In addition to those more abstract concerns, [111] also notes a potential loss of efficiency, compared to hand-written software.

3. Related Work

As there is a lot of literature regarding QoS monitoring and the specification of QoS or SLAs, there are lots of different architectures. This chapter tries to summarize current work in this area and its relation to the approach of this thesis. The first section discusses general works on monitoring and management architectures, including some specialized on quality classes in SLAs. The second section is about approaches, emphasizing special QoS parameters. It covers general and semantic approaches as well as custom QoS metrics using temporal evaluation or the monitoring of custom QoS metrics in service compositions.

3.1 Monitoring

This section discusses related work in the field of QoS monitoring. In addition to general QoS monitoring solutions and their architectures, works using quality levels and extended management frameworks are described.

In [115] general requirements for a Web service monitoring system are defined. A central part of the description is a `quality model`. The paper notes that state changes as well as other events influence the quality of services. It identifies five classes, which should be monitored: *Response Message*, *Application Execution* (e.g., wrong business logic, inconsistent data, memory leaks or deadlocks), *Resource State Changing*, *Request Message* and *Management Operation*). Management Operations, such as re-configuration or resource adding, should be measured to evaluate their correctness and effectiveness. The proposed measurement architecture consists mainly of a `central analyzer`, which is able to check constraints, predict risks and find problems, a `high level representation`, as well as `probes` and `agents`. The `high-level representation` visualizes the analyzed results, such as violated constraints, current risks, the reason for the violation or risk and, potentially, a possible solution, for administrators. `Agents` and `probes` are used to measure the concrete values, which are used by the `central analyzer`. `Agents` are run in a separate process from the service and extract information actively through API calls. `Probes` are embedded into the same process as the services and extract events passively. They can be either be added through instrumentation or via interceptors. Another design decision is how much processing is done by the probe. A `probe` can just capture events and relay them to the analyzer, or it can already analyze the events and only send reports. The presented prototype is based on *Apache Axis* [100]. The communication between `agents`, `probes` and the `central analyzer` is implemented using Java remote method invocation (RMI) [80] for simplicity reasons. An evaluation led to the result that instrumented probes provide the least deviation from the exact result and that the separate monitor adds a performance penalty. While [115] uses probes and agents for measuring of QoS, we propose the use of a framework which adds support for different MPs to the monitored services. This allows for a greater flexibility, is easily extensible for new measurements and supports application-specific QoS out of the box. Our current prototype does not support monitoring of management operations, but this feature can be added through additional MPs.

Besides providing a short overview over some monitoring tools, [10] provides a QoS model and and a monitoring solution. It aims to enhance server-side monitoring by providing a rich, but not complete, server-side QoS model. The six defined QoS attributes are: *availability* - The probability that the service is available at a given time, *accessibility* - How many of the requests have been validly responded to, in relation to all requests, *performance* - Throughput and Latency of a service, *reliability* - How well a service can maintain service quality, *security* - How vulnerable a service is, measured by searching in vulnerability databases for impact scores and *regulatory* - How well a service conforms to rules, laws, standards, etc. Additional attributes, such as robustness, accuracy and reputation are mentioned for future consideration. The paper also presents an implementation of a server-side monitoring tool using Java Management Extensions (JMX) [51]. This prototype uses the Java system application server (now called Glassfish [46]) to build on top of its basic monitoring support and integrates itself into the management console. A major difference between this approach and the one presented in this thesis is, that the approach in this thesis does not use predefined metrics, but allows for custom definitions of standard attributes. It also supports metrics at the client-side and possibly other locations (e.g., SOAP intermediaries - see Section 2.2.1) if corresponding MPs are created.

[110] presents an extended taxonomy of QoS metrics as well as a peer-to-peer based monitoring framework for Grid applications. The main classes of the taxonomy are *performance*, *dependability* (security, availability, accuracy, etc.), *configuration*, *cost* and *custom metrics* (service specific measures). Often there are different ways of measuring a single metric (e.g., reliability or availability can be measured through Mean Time Between Failures (MTBF) or Mean Time To Repair (MTTR)). As there are a lot of QoS metrics, not each one can and should be measured at the same time. A certain selection should occur. The architecture is based on four components: `sensors`, `middleware`, a `client user interface (UI)` and a `QoS Knowledge base`. The `sensors` collect data from resources and send it to the `middleware`. Each resource has a unique ID and type (machine, network path, middleware or application). The middleware in the prototype is based on SCALEA-G [92], a peer-to-peer infrastructure for monitoring of Grid services. A `QoS Knowledge base` contains analysis rules for metrics and resources as well as dependencies between those resources and historical data from previous analyses. The `client UI` includes a GUI and a reasoning engine to reason about the collected QoS data. This engine also allows for automatic rules which enable the system react to system changes. In contrast to our approach, this approach uses peer-to-peer monitoring and a fixed taxonomy. Our approach uses a more flexible measurement description DSL and a centralized server. While the peer-to-peer solution promises better scalability, in our approach one server could be used for each service, thus improving performance on heavy loads while having less communications overhead. Similar ideas are, that there are lots of metrics which should only be measured when necessary and that even generally accepted metrics can be measured in different ways. Our solution directly approaches these issues by assigning only needed MPs to the services and by presenting the MDDSL for unambiguous definition of QoS metrics.

[103] separates *provider-centric* and *client-centric* measure collection. Some measurements can be collected at both points (e.g., performance, availability, reliability), but some only at the provider or client. Provider-centric attributes are generally close to the implementation of the service. Client-centric attributes are mentioned as an important addition. Both types of attributes are sent to a central `QoS Measure Updater` which updates the QoS information in a registry. Measurement on both the client and provider side can be implemented through *low-level packet monitoring*, the use of a *proxy* as a mediator or *SOAP engine library modification* (i.e., modification of the library to emit necessary information). Some provider-centric attributes must be monitored manually (e.g., security, accuracy and precision). A prototype is introduced, which uses the *library modification* approach on *Apache Axis* [100]. It uses separation of measurement collection and sending to reduce the influence of measurement during peak times and to foster reuse of the individual components. Similar to our approach, [103] uses a centralized monitor and supports different measurement methods at different locations. It also uses a separation of measurement collection and sending to improve performance. Our solution allows to measure the same properties without relying on fixed definitions, but by using metrics defined in the MDDSL.

3.1.1 Quality of Service Levels

Some frameworks support different levels of service quality for one service. This enables service providers to change their QoS intentionally, based on the selected quality class. That way, higher paying customers can get higher service quality, less influenced by other customers.

The WS-QoS [104, 105] framework implements service monitoring and service selection. It is possible to monitor predefined or custom QoS properties. It allows to define SLAs using `offers` and `requirements` for services as well as quality classes of services. Offers represent the least provided QoS, while requirements represent the least needed QoS. A service can provide different guarantees when used on a different quality level. The architecture is built of the basic SOA triangle (see Figure 2.1(a)) with an additional broker. Instead of a registry, the broker is used by the requestors to acquire a reference to a Web service, which implements a certain interface and offers the needed QoS. In addition to the broker, there is a `Requirement Manager` and a `Monitor`. The `Requirement Manager` stores clients QoS requirements. The `Monitor` checks for the compliance of service offers. It also provides a graphical overview over requirements and offers and can also be used to monitor headers of SOAP messages. [104] emphasizes the network layer QoS support. `Proxies` are used by requestors and providers to support certain requirements. As QoS information is sent in the header of the SOAP messages, the `proxies` can be used to add support for QoS in the network layer by mapping the requirements to the lower level. To support custom QoS properties, an editor is implemented. It allows to define an ontology for each added property. Compared to our solution there are not many similarities. It uses quality classes and, in addition to offers of a service (promised constraints), it supports requirements, which are enforced by the used proxies. The sending of QoS information in the headers allows those proxies to set a certain required network quality level, but also adds an overhead to the measurement. In addition there is an overhead of calling the proxy before each invocation, which also does not exist in our approach.

3.1.2 QoS Management

The following approaches use monitoring as a part of an management framework. This allows to view monitoring in an extended context. The monitored data can be used for SLA violation detection (including a wide set of penalties to enforce the constraints), selection of services, adaptation, reputation mechanisms, etc.

The CosmosQoS [62] framework is used for QoS evaluation and service selection. It uses a proxy-based approach for measuring the QoS of services and supports a reputation appraisal mechanism. The presented architecture is based on two types of proxies: `regional proxies`, and an `universal proxy`. `Regional proxies` reside near the location of the clients, at a networks gateway or proxy, and measure and store QoS information. The `universal proxy` frequently queries the `regional proxies` to provide a more general overview of all measured QoS properties. Based on the discrepancy between the price a requestor is willing to pay and the price of the invocation and based on the deviation between the required QoS values and the historical credibility, a score for each service can be calculated. This score is used for matchmaking. It can be influenced by putting weights on the price, QoS deviation, historical credibility and the QoS properties. While our prototype uses interceptors and the solution also supports other MPs, the CosmosQoS framework only uses proxies as data sources. Support for such proxies could be introduced by adding additional MPs, but [79](in Figure 4.27) shows superior performance of interceptors to proxies. Another benefit of using interceptors is the simpler infrastructure of the service. Our solution does not include evaluation of a service for a concrete purposes but only measures the QoS values. A matchmaking component would be a separate component in an extension of our architecture. Historical credibility can be added by using a new MPs, which fires in case of SLA violations.

In [87], an overview of ways to monitor QoS and describe SLAs is presented as well as a system for penalties. The proposed approach uses WS-Agreement [5] as a foundation. A SLA can be used to constrain providers or clients, and typically contains SLOs. According to [87], those SLOs should have a service quality level associated with them. Another important part of SLAs are the penalty specifications. Penalties can be necessary if a single SLO is violated, if some specific SLOs are violated or based on weighting of the SLOs. The penalties themselves can be weighted based on the severity of the violation. Concerning monitoring, the paper advertises the use of trusted third party monitors (external to the service) or trusted third party modules (internal to the service). It states that those solutions are superior to requestor-only monitoring, as a requestor can never prove by itself that the violation really occurred. The paper also strongly promotes monetary penalties as they supposedly lead to a better overall QoS for both, service providers and clients [87]. Instead of using WS-Agreement and quality levels, our solution uses a separate and simple SLA language, tailored for better support of the different stakeholders [79]. Similar to [87], our approach also promotes a third party monitor and a third party module. An interesting idea is the usage of SLO weighting and its influence on penalties. This could, combined with different quality levels, provide a good extension to our current solution.

In contrast to our approach, [57] advertises using client feedback for monitoring. It highlights the trustability and performance problems of trusted monitors, provider-side monitoring and active probing to press the point. While the proposed solution does not try to make dishonest client reports impossible, it argues that cheating is made uninteresting. To encourage honest reporting of service quality attributes, a payment mechanism is proposed, which depends on the vote of other clients and makes telling the truth the optimal solution for any client. While this only works if a fraction of clients report truthfully, this is assumed and enforced by using external trustful monitors. Additionally, collusion between clients is prevented by adding incentives for leaving malevolent coalitions. Client feedback is an interesting and complex topic. While our solution does support client feedback through user defined application-specific QoS metrics, it is at the moment only implemented rudimentarily. As many challenges appear when using client feedback and trying to prevent cheating, [57] provides a feasible approach which can be fit to our solution.

[116] uses a special *Arbiter* component for adaptation support. It introduces an SLA model and a QoS model and demonstrates an architecture, called *SLAMon*, for monitoring SLAs, using a predefined set of QoS properties. The basic architecture of the approach includes `Measurers`, a `Monitor`, an `Analyzer` and an `Arbiter`. The `Monitor` calls the `Measurers` regularly to receive current measurements, which it stores in a database. The `Measurers` are created by the monitored services. They support different kinds of services, such as Web service, HTTP services or agent-based services [116]. The `Analyzer` checks for violations of the SLA. If a violation is detected, it notifies the `Arbiter`, which can take appropriate actions. Alternative services, which can be chosen, or other actions could be stored in a repository for automatic correction. Similar to our solution, the `Measurers` are partly embedded into the services and provide some technology-independence. As in our approach, adaptation would be separated from the concrete monitoring subsystem. `Measurers` in [116] are polled by the `Monitor`, which prevents timely updates and violation detection if not compromising performance by polling frequently. Besides using user-defined QoS properties, the active notifications *of the monitor* by the measurement components, is one of the advantages of our system.

The *Vienna Runtime Environment for Service-oriented Computing (VRESCo)* framework is introduced in [74]. It aims to tackle current challenges in SOC. According to [74], the framework supports, among others, service composition, dynamic binding (and rebinding), dynamic invocation, searching, querying and notifications. Dynamic binding can be based on content of the service or on QoS. It also allows various rebinding strategies. The framework is separated into various services [74]:

- The `Publishing and Metadata Service` stores interface description and metadata of a service. Metadata incorporates functional as well as non-functional attributes. The VRESCo framework does not use a separate registry but implements its own.

- The `Searching and Querying Service` implements searching for services. A special query language was developed to support metadata querying as well as service selection [73].

- The `Binding and Invocation Service` is used for dynamic binding. It is used through a client library.

- The `Notification Service` supports notification of internal as well as external consumers [71]. Its function is explained later in the text.
- The `Composition Service`. VRESCo has a separate orchestration language which offers several of the features of the framework.

The monitor from [91] is used to monitor predefined QoS attributes. The monitored attributes are measured by active probing [91]. In addition there are monitors at the service location [72] which measure real service calls. Methods to evaluate a service include *WSDL analysis*, *TCP packet capturing* and *active probing* (e.g., for throughput). The measuring implementation is weaved into the client implementation using aspect-oriented programming [91]. Events from the monitor are sent into VRESCo's `Notification System` as external events and stored in the registry. The Notification System also supports a variety of other events (e.g., for management, querying or versioning). It is based on *NEsper*, a port of *Esper* [31](See Section 4.2) to *.NET*. SLA obligations are translated to Esper Event Processing Language (EPL) [32](see Subsection 4.2.1), so violations can be easily checked [72]. Notifications are not only used internally, but can be subscribed to and sent externally via WS-Eventing [12]. A more detailed description of the eventing system can be found in [71]. As the VRESCo framework provides a complex management solution, only the monitoring part is comparable to our approach. In [91], services are measured using active probing instead of monitoring, thus testing the service actively in an *evaluation phase*. Our approach does not actively invoke, but observe the service during its operation. Similar to our approach, the VRESCo framework is also heavily event-based, showing what flexibility event-based architectures allow, by providing a complete management solution. It includes support for message invocations, leading to a tighter coupling of the application to the framework. Our approach tries to minimize impact on the existing system by using non-invasive techniques like interceptors and by not interfering with the way the used Web service middleware operates. A big advantage of our system is the custom specification of QoS metrics using a DSL and their incorporation into the event-based system.

3.2 Extended QoS Metrics

This section describes some relevant works which include some support for custom specification of measurements. At first some general frameworks and approaches are presented, then different detailing approaches, such as semantic approaches using ontologies, explicit inclusion of time and monitoring of QoS compositions are showcased. As nearly every paper on the subject also discusses a measuring framework, the architecture for each is also described briefly.

IBM [48] developed the Web Service Level Agreements (WSLA) framework [59]. It is used to specify and monitor SLAs, including the description of the used QoS properties. A WSLA specifications are XML-based. They include the *involved parties*, *service descriptions* and *obligations* and references the WSDL document. The service descriptions define the SLA parameters (e.g., QoS values) which should be measured and references a *metric*. *Composite* metrics are composed of other composite metrics or atomic (*resource*) metrics. A composite metrics has a corresponding function with other metrics as parameters. A resource metric has a measurement directive which defines how it should be measured, by specifying a source and the method of retrieval (push or pull). Obligations consist of *SLOs* (guarantees and the corresponding obligor) and *action guarantees*. An action guarantee uses a reference to an SLO as a precondition. If this condition is fulfilled, the corresponding *action* is taken. An important part of the runtime architecture, which is used for the evaluation of SLAs, is the *SLA Compliance Monitor*. It consists of three services. The `Deployment Service` is used to control the lifecycle of an SLA and to split it into relevant parts for the other services. The `Measurement Services` uses the metric definitions and supports a variety of data providers, which add support for specific measurement implementations. The `Condition Evaluation Service` receives metric updates from the `Measurement Services` and checks for condition violations. If a condition is violated, the architecture allows to take corrective actions through a `Management Service`, which proposes actions to the `Business Entity`. This `Business Entity` internally represents business goals, policies and general knowledge and can approve to the action or even propose alternatives. While the goal of this approach is similar to our approach, even though the focus is on the measuring aspect, there are some important differences. The WSLA framework uses an XML-based language for a very detailed SLA. Our solution splits the measurement description and the SLA into separate documents. This, and the usage of a external DSL, allows the SLA itself to be simplified for better support of stakeholders with different backgrounds. A similarity to our solution lies in the splitting of the document through a deployment service for separation of concerns. Resource metrics can be roughly mapped to MPs, while any metric used in an SLA is some kind of composite metric.

[42] proposes a framework for monitoring observable metrics based on a peer-to-peer architecture, which extends the distributed QoS registry *Q-Peer* [61]. Every peer has a QoS monitor, which consists of four modules, each in turn consisting of a number of "logical" processors: The `data collector` filters information from the SOAP messages. The `metric generator` generates the metric values from actual measured data, from a data collector or an external monitor, combined with historic data. The `feedback controller` evaluates policies to detect deviations in the services QoS and can subsequently take actions to limit the impact of the policy violation. The `configuration interface` is used to feed those policies into the system. A monitoring policy consists of a *trigger* and an *action*. It is decomposed into a *data collection policy*, which defines how data should be measured and is send to the logical data collectors, and a *metric generation policy*, which is specified in WS-Policy [112] and is sent to the logical metric generators. The architecture supports measuring of single services ("single-party monitoring") or multiple services ("multi-party monitoring") for one metric. It also allows to calculate metric values from other metrics ("metric regeneration"). The most obvious difference between the approach from [42] and our approach is its use of a peer-to-peer architecture while our approach uses a centralized monitor. Our solution does not support multi-party monitoring and metrics are not built based on other metrics, but always on MPs. Reuse currently happens at the transformation level instead of reusing other metrics. A big similarity lies in the separation of data collection policy and metric generation policy, which corresponds to MPs and the transformations based on Measurement Descriptions (MDs). Advantages of our solution are the use of DSLs for stakeholder support and its symbiosis with the event-based architecture inside the monitor.

In [114], a management system for Web services is presented, which supports contract establishment, monitoring, diagnosis and adaptation. When a contract is established, the `Monitoring Service` is responsible for the monitoring part. There are two types of agents for monitoring: `Monitoring agents` receive information actively from resource monitors. `Probing agents` are used to poll resource data regularly or on-demand. Both agent types sent the measurements to the *Diagnostic Service*. Using causal networks, it aggregates the received data and is able to react to contract violations, to determine the reasons for those violations and to notify other components. A separate `Adaptation Service` uses XML-based adaptation specifications to react if a contract violation occurs. [113] goes into a bit more detail concerning the systems QoS specification. It supports a number of predefined characteristics, such as availability, message ordering and information accuracy. To specify a QoS attribute, its characteristics, value domain(nominal/ordinal/numeric) and range, and constraints, which describe its relation to other attributes, are defined. One difference to our solution is that the aggregation of values happens in a separate diagnostic service. While our monitoring system uses event processing, the solution from [113] uses causal networks and includes diagnostics. This could be enabled in our system by adding additional attributes to the sent events. Another difference is that our approach only uses passive monitoring, storing all information in the monitor. Instead of polling a probing agent, value updates are sent to the monitor as they happen. The monitor can then be queried. At the moment our prototype only supports numeric data, but support for textual data could be implemented. In addition to providing a DSL for the specification of measurements, our approach also allows updates at runtime using a distribution system.

3.2.1 Semantic Approaches

Many approaches use ontologies as a means of knowledge representation. This allows for a better cooperation between different systems as they can share ontologies or use translators between those ontologies to simplify communication of concepts. It also provides a good overview of things to consider when implementing a similar system.

Some works provide ontologies for (semantic) Web services. The Web Service Modeling Ontology (WSMO) [22] is such an effort. It can be used in combination with the Web Service Modeling Language (WSML) [23], which is providing formal syntax and semantics. While [106] extends WSMO by adding additional ontologies for modelling QoS characteristics, it does not add a detailed description of *how* the measurement should be done. Nevertheless, ontologies provide common concepts and including references to those ontologies might be an interesting extension.

The Web Service Offerings Language (WSOL) [109] is is an XML-based language for specifying SLAs. It uses the idea of *classes* of services. *Offerings* are the formal representation of such a class. This allows WSOL to define constraints on QoS, functional properties, payment, simple access rights, etc. Constraints on the operation of the service are defined through boolean or arithmetic expressions. An offering can be scoped on the port- or service-level. While WSOL offerings use ports and services from WSDL, they use other constraints and are therefore kept in a separate file. WSOL uses external monitoring sources via a `managementResponsibility` construct, but the constraints themselves are expressed in the WSOL specification. Alternative services, to use in case a constraint is violated, can be defined. [108] describes the language features in more detail, for example, *service offerings and their relationships*, *service constraints*, *management statements* and *reusability constructs*. It also discusses some applications of the framework for Web service management and composition and mentions that WSOL needs additional, external ontologies of QoS metrics. At the moment it is assumed, that these ontologies just list data types and units corresponding to names, later is is planned to include a more complete set of ontologies. An effort to define common requirements for such ontologies is defined in [107], while emphasizing the need for shared QoS ontologies. It proposes five different ontologies focusing on definitions of metrics, currency units and measurement units. To describe a metric it uses a *name*, a *textual description* for humans, the *measured property* (e.g., time, quantity of information) based on another ontology, *formulas for metric computation* and *invariant relationships to other metrics*. The measurement unit ontologies includes base units (e.g., seconds) as well as multiplicities of units (e.g., milliseconds), derived units (e.g., invocations per second) and other concepts such as synonyms, abbreviations and conversion rules between those metrics. This solution provides complex offerings, not only based on QoS but including other attributes. Our approach concentrates on the measuring of service quality. Instead of using an XML configuration, our approach uses DSLs, specifically tailored to different stakeholders, improving usability. A possible future extension for our system would be a complete unit ontology, as it is presented in [107]. This would, among other benefits, aid in automatic unit conversions. As [109] only specifies alternative services in case of SLA violations, the possible reactions are very limited. We suggest a separate adaptation component (like in [116]), which reacts on notifications from the monitor (see Subsection 5.2.2). According to [108], WSOL also provides some support for dynamically managing the current offers of a service and the relationships between those offers. These so-called SOM ports serve a similar purpose as the distribution system in our approach.

[68] proposes an ontology for QoS and an agent-based architecture, which uses this ontology for service selection. The agents act as autonomous brokers for service consumers. They share the QoS information among each other using agencies. If a consumer wants to bind to a service, he starts an `agent proxy` and provides his requirements via an XML policy language. The agent uses the agencies and other agents to determine the best service. It also acts as a proxy for the invocations, monitoring them in the process. An agent can also acquire feedback from the user. Knowledge representation is implemented via two main ontologies. The *Service Ontology* is used to assign services to domains and to associate qualities with those domains. In addition it can represent agents and their behaviors. The *QoS Ontology* is separated into three sub-ontologies. The *Upper QoS Ontology* describes commonly used properties and relationships between QoS properties. The *Middle QoS Ontology* represents QoS properties ("quality aspects") in a hierarchical manner. It is supplemented by a *Lower QoS Ontology* which is domain-specific. The agents in this approach act similar to a combination of broker and proxies in other works, therefore every call has an additional overhead for the call to the agent. In our system, a proxy can be used, but the current prototype uses interceptors which has a performance benefit [79](pages 64-65). The system of [68] is strongly based on knowledge representation, using a variety of ontologies. Those ontologies also include predefined QoS attributes in the *Middle QoS Ontology*, but allowing extra domain-specific attributes using

another ontology. Our solution also allows to define new attributes, but includes the concrete measurement process using predefined functions.

As there is a variety of ontologies available for SLA and QoS specification, it is not easy to choose a fitting ontology. Interoperability can not be guaranteed without a common ontology or transformations. [25] strives to unify those ontologies by providing a common set of requirements. It lists current solutions, highlighting their shortcomings, disadvantages and advantages and provides concrete requirements for different scenarios. Requirements are listed for *general ontologies in this domain*, *QoS concepts*, *unit and conversion ontologies* and *SLAs and SLA concepts*. By providing those common requirements, this work can aid in unifying the abilities of QoS monitoring systems. While some of the requirements are deliberately left out because they are out of the scope of this thesis, some of them, such as unit conversion, would definitely be worth including in future versions.

3.2.2 Temporal QoS Parameter Evaluation

To be able to measure some QoS parameter values, it is necessary to look on the temporal relationship of different measured events. Some approaches directly allow to specify QoS metrics using time.

[86], for example, uses timed automatons to describe complex QoS metrics. It builds on a *SLAng* [95], which allows to define SLAs by defining constraints for services as well as their clients. The constraints are transformed into timed automatons, which accept a language representing violations of the constraints. In a presented prototype, events are collected via Axis [100] Handlers and checked for violations using `Checkers` at the client and monitor. An evaluation in the paper shows that this method has only little overhead, is non-intrusive and can be implemented easily into existing applications. Like our approach this approach tries to be non-intrusive. Instead of EPL it uses temporal logic. By using DSLs, our approach tries to be easier to understand and to extend. In our system, SLOs are also negated and transformed into another language for processing, but it supports additional measurements, for example, availability-related measurements and general application-specific QoS measurements.

EVEREST(EVEnt REaSoning Toolkit) [96] is the monitoring framework of the *SERENITY* [94] project. It uses EC-Assertion, an XML-based temporal logic language based on event-calculus, to check for violations of SLAs. `Event capturers` intercept events of the application and sends them to a `monitor manager`. The `monitor manager` is used to coordinate the measuring process. It sends the messages to a `monitor`, which checks them against the EC rules with simultaneous consideration of the system state. QoS properties are also expressed via EC formulas (e.g., MTTR in [63]). Two challenges of this approach are time differences of the machines comprising the system and the required monitoring lifetime of events [96]. To allow to calculation of timing differences between two locations in a network it uses clock synchronization. An algorithm is proposed to determine the necessary event lifetime. *EVEREST+* adds additional *SLA violation prediction* for the *EVEREST* framework [63]. *Prediction specifications* allow to implement user-defined `QoS predictors` with different algorithms. They are transformed to EC formulas by the `Monitoring Specification Generator`. The event capturers roughly correspond to our client- and service-framework and MPs. The challenges of time differences between multiple services are considered by using clock synchronization. Our solution does not address this, but assumes unsynchronized clocks. It is, however, discouraged but possible to compare timestamps from different systems. Concerning monitoring lifetime, EPL provides direct support for time-constructs (such as timers or timeouts), simplifying the managing of event lifetime. Thus timeouts can be set when creating the MD.

[20] uses *EVEREST* as its default monitor. The approach presented uses a management framework defined in the *SLA@SOI Project*. It is based on the assumption, that there are complex SLA hierarchies with dynamic SLAs, that can change, be renegotiated, etc. Besides monitoring, the paper also heavily discusses SLA negotiation and its connection with monitoring. The paper states that it is not reasonable to assume that QoS properties can be calculated without historic data and that it also does not make sense to negotiate an SLA when it can not be confirmed that the monitoring system can monitor the properties. An architecture is proposed to monitor the fulfilment of SLA contracts. It consists of `Event Captors`, an `Event Bus`, a `Monitoring Terms Derivation Module`, a `Terms Verification Module` and a `Monitoring Engine`. `Event Captors` have different implementations. They are used to monitor resources or system states. The `Event Bus` implements a publish/subscribe architecture. It is used to transmit events from the captors to monitors. The `Monitoring Terms Derivation Module` translates SLAs into measurement terms in the form of assertions written in *EC-Assertions*. The `Terms Verification Module` tests if the SLA is monitorable by checking the pre-exchanged capabilities of the system. The `Monitoring Engine` receives events, processes them and stores violations as historic data in a database for future negotiations. As this solution builds on the *EVEREST* framework like [96], the relation to our work is very similar. Some interesting aspects of this work include the SLA negotiation and an emphasis on historic data, which would complement our monitoring system on the way to a management system.

3.2.3 Monitoring Compositions

As it is important to monitor individual services in service compositions to be able to estimate the resulting QoS of the composition, a lot of research focuses on monitoring compositions. Although the following works use different approaches to monitor compositions of web services, the task of providing custom QoS attributes for composition is very similar to that of atomic services.

[16] describes a method of generating a monitoring system from an SLA and a business process. Based on BPEL, certain events in a workflow are emitted, collected and processed. The proposed SLA structure is built on and extends ideas from WSLA [49]. The approach allows to define SLOs and metrics in a reusable way, through a typing system. SLOs are monitored and if a violation is detected, action handlers are called. A SLO is built from different metrics. Metrics are defined using `event handlers` and are allowed to have parameters. The events are spawned from the business process. The concrete monitoring system consists of `monitoring clients`. For every metric, SLO and action handler, such a client is generated and registered into the system. It is then called based on a publish-subscribe mechanism. In contrast to our

work, [16] uses events which are sent from a workflow. Like [42] it reuses metrics, which our approach does not do. An advantage of this system is that created metrics can have parameters, which would be an interesting extension to our prototype (see 8.1).

A *model-driven* approach to create the monitoring infrastructure for a Web service composition is introduced in [76]. Separation into multiple abstraction layers makes it possible to provide a high level of platform independence. As the framework supports multiple interconnected metamodels, it is possible to define the points in a Web service composition at which values should be measured, the methods for calculating metrics as well as the *indicators* (which represent metrics) using those methods. These methods are called `calculation templates` and provide a way to reuse operations. The metric value is updated based on a `update rule` which is also included in the indicator definition. To allow to measure a concrete value, the model has to be transformed to a platform-specific model. Besides composition monitoring, another difference between [76] and our work is, that our system does not generate all of the monitoring infrastructure beforehand. Instead, we use a fixed but flexible infrastructure, which allows changing of measured properties at runtime, while maintaining similar domain-independence. The calculation templates correspond to the functions defined in the MDDSL, the indicators are similar to our MPs.

[118] describes a way to measure QoS values in compositions and what should be considered when measuring them. For that purpose, it proposes a method for metric computation and an architecture for monitoring *observable* metrics. According to [118], challenges for such an architecture include: a high number of possible events which are registered, complex relations between the metrics and the storage of the events. To argue for a system which allows explicit metric generation, the paper states that a standard QoS model for Web service in all domains is not practical. A metamodel for QoS metrics is presented. To determine where the measuring should take place, a workflow can be imported and an observation model is generated. For every service execution an `execution activation event definition` and an `execution completion event definition` are generated. The computational logic is implemented through Event-Condition-Action (ECA) rules, based on those events. The framework brings an own event computation engine. It uses state-based models with a mixture of compilation and interpretation and includes a method for execution planning. For persistence it uses a relational database. Both [118] and our approach assume that a standard QoS model could not fulfill the needs of stakeholders, making explicit metric generation a must. An observation model is extracted via workflow analysis, which could be used as a base for MPs in our system, effectively extending it to a composition monitoring system. While this system uses ECA rules and an own event computation engine, we use a standard and well tested CEP solution, which not only provides advanced time- and pattern-constructs but also allows for future extensibility through incorporation of new event types. Those new event types can even be queries through new MPs, providing new possibilities to the metric designer.

In [11], two formerly separate approaches for composition monitoring, *Astro* and *Dynamo*, are joined. Dynamo focuses on single processes, using Aspect-oriented Programming (AOP) and the Web Service Constraint Language (WSCol) language to extract information from BPEL processes. It allows to define properties with a very high granularity. Astro has a much higher granularity. It uses a language called Run-Time Monitor Specification Language (RTML) to aggregate events, which spawned from message exchanges, to complex events. RTML is based on linear temporal logic. Through combining the two approaches, a highly flexible system is generated. WSCol is used as a base language for defining *basic events*, which are defined through their declaration, location (via XPath [18]) and runtime parameters. It is managed through the `Basic Events Manager`. *RTML* is embedded into an external `Composite Properties Monitor` and using linear temporal logic to calculate properties from basic events using aggregation formulas, etc. Basically, it supports two kinds of properties: `Instance properties` are defined per process instance and `process properties` are defined per process. Although it is used for monitoring services compositions, this combined approach is very similar to our approach. Basic events correspond roughly to our MPs. Specifications in WSCol would provide a good MP for composition monitoring. The RTML corresponds to the MDDSL. Both, our and this, solutions use events, but while [11] uses them for sending basic events, our solution uses a CEP system for the transformation to complex events. This allows to reuse complex events as basic events (e.g., for measuring the average SLA fulfillment) or for integrating a penalty system directly into the event system. Another benefit of our solution, compared to [11], is the ability to change measurements at runtime. The distribution management system automatically notifies the clients of updates, enabled by the capabilities of the frameworks at the client- and service-location.

In contrast to the prior works, [14] focuses on *application-specific* QoS attributes, such as color-depth or refresh rate. For those attributes its not possible to determine automatically how the QoS values of individual services aggregate to global QoS values. Therefore, for every attribute, aggregation formulas have to be defined in addition to the basic specification. The approach of [14] specifies a data type and a scale for each attribute. Included scales are ordinal, interval, ratio or absolute. Aggregation formulas are defined for each combination of attribute and workflow construct in a language which borrows heavily from the Object Constraint Language (OCL) (a language for expressing constraints on Unified Modeling Language (UML) models). A `QoS Aggregation Tool` is presented which includes an `aggregation function editor`, a `type checker` for verifying the type system and a `formula interpreter`, which evaluates the global QoS of a workflow. The ideal services for the composition are determined at runtime. The workflow uses proxies for each service, which allow dynamic binding and rebinding and act as "abstract services" [14]. Before binding to a service, the proxy chooses appropriate services and sends them to a `Binder` component. The `Binder` optimizes a fitness function for the workflow using genetic algorithms and returns a near optimal solution to the proxy, which can then bind to the service Using such a system when composing services, even the application-specific QoS of the composed service can be estimated based on the used services. As every attribute in our system is defined by the user, no assumptions can be made how they could be aggregated in different workflow constructs. Attributes, which enable this, could be easily added to the MDDSL in future versions. This would enable a binding component to find fitting services to reach the best possible overall QoS.

4. Background

This section describes the background of this thesis, consisting of the used frameworks and the work it is based upon. The first section describes *Apache CXF* [6], the framework used for creating the Web service interfaces. It is also the framework on which the QoS of the Web services is measured. The second section describes *Esper* [31], which is the used CEP framework. In the third section the *Frag* [101] programming language is explained, which is used for implementing the DSLs. The last section describes the *QuaLa*, which this thesis is built upon. It is a set of DSLs and accompanied by an architecture for the measurement of the QoS of Web services.

4.1 Apache CXF

Apache CXF [6] is a framework for building services based on a variety of technologies. It supports many WS-* standards such as WS-Addressing, WS-Security, etc. As Apache CXF is a large, flexible and complex framework, and there are always many ways to do things, only the parts which are necessary for this thesis are explained. Its documentation can be found in [7].

CXF provides a very flexible structure with a wide support of technologies. *Data Bindings* describe how XML elements are mapped to Java objects and vice versa. *Protocol Bindings* map formats and protocols onto transport technologies (e.g., SOAP, see Subsection 2.2.1). *Transports* are used to hide transport protocol specific details from the other parts of the system. Different transports include support for the HTTP, JMS, etc.

4.1.1 Message Processing

When a client or service sends or receives a message, an interceptor chain is created. Every interceptor in a chain is called in a specific order, through their `handleMessage`-method.

If the message exchange is a request to a service, the client creates an outgoing chain and the service creates an incoming chain. If the service replies, it creates an outgoing chain and the client creates an incoming chain.

Interceptors build the basis for message processing in CXF. They can modify messages (including headers), add to the messages, validate their content and/or throw faults. Chains are split into various phases (e.g., `Receive`, `Unmarshal`, `Send` or `Invoke`) to impose a certain ordering. Inside each phase, the interceptors can be specified to be before of or after certain other interceptors. Figure 4.1 shows exemplary calling orders of the interceptors in the chains. The green bars are outgoing chains, the whites bar incoming chains and the red bars are fault chains. The grey box symbolizes the invocation interceptor of the system, which invokes the service implementation at the service's CXF framework. It shows that the calling order of the custom

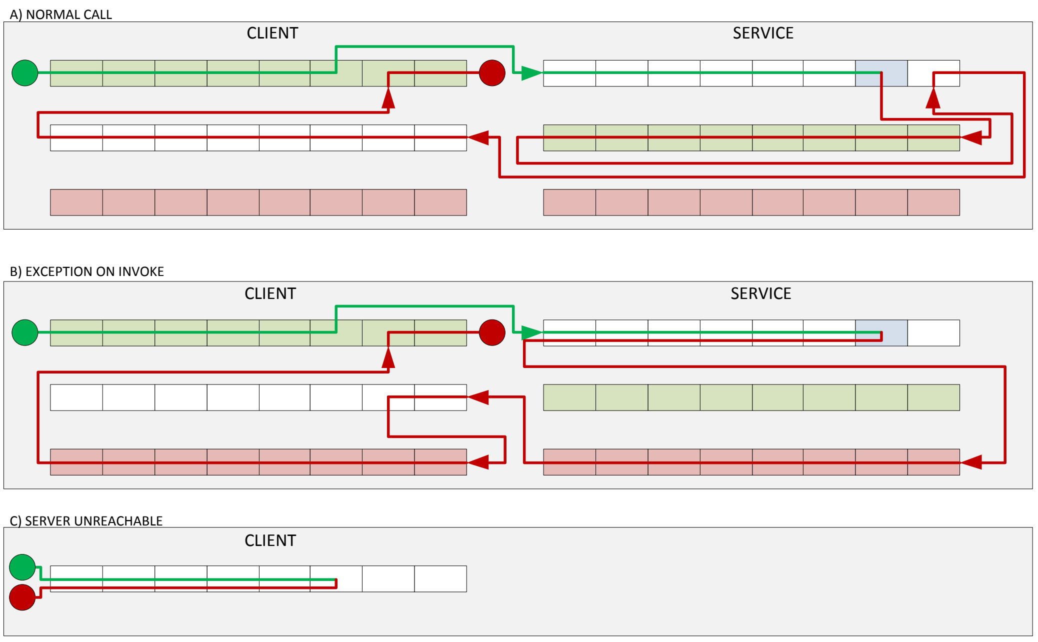


Figure 4.1: Exemplary Calling Orders of the Interceptors

interceptors is strongly dependent on the call, as the sending of the message and the execution of other chains are also executed by system interceptors.

If a fault is detected while processing a chain, it is unwrapped. This means that all called interceptors get called in the reverse order on their *handleFault*-method. A special case takes effect if a SOAP fault occurs. After unwrapping the respective chains, the service starts an extra outgoing and the client an extra incoming fault chain which are used to send or receive the corresponding SOAP fault messages.

4.1.2 Building Web Services using CXF

Web services can be built in a WSDL-first or Java-first way. Using WSDL-first, Java code is generated based on a given WSDL file. Additional code can be added afterwards. Using Java-first, Java code is annotated with JAX-WS [50] annotations, so a WSDL file can be generated. This thesis uses the Java-first way.

Services

To build a Web service from a predefined Java interface, all that is needed are four annotations (see Listing 4.1). The `WebService` annotation marks the interface as a Web service, the others are only necessary in case of ambiguities (e.g., identical function names but different parameters)

```
@javax.jws.WebService(name="EchoService",
    targetNamespace="http://happens.at/EchoService")
public interface IEcho
{
  @javax.jws.WebMethod(operationName="echo")
  public @javax.jws.WebResult(name="echoReturn") String
      login(@WebParam(name="echoParam") String echoParam);
}
```

Listing 4.1: An Example JAX-WS Annotated Web Service Interface

```
JaxWsProxyFactoryBean clientFactory=new JaxWsProxyFactoryBean();
clientFactory.setServiceClass(IEcho.class);
clientFactory.setAddress("http://localhost:9000/EchoService");
IEcho client=(IEcho)clientFactory.create();
String echoText=client.echo("Test");
```

Listing 4.2: Invocation of a Service

or to configure the conversion to WSDL. The `WebService` annotation is used to define a name and a target namespace to the service, represented by the interface. The `WebMethod` describes the name of a method in the WSDL. The `WebResult` and `WebParam` annotations, respectively, describe the name of a return type and of a parameter of a method.

To start such a Web service, there are factory objects, which can be used to create a service based on a service implementation ("serviceBean"), an interface("service class") and an address.

Clients

One way to build a client in CXF is through a `ProxyFactory`. It can create a proxy for an annotated interface (as it was used to describe the service). Only an additional address attribute is needed. An example for a client creation and invocation is shown in Listing 4.2.

Adding Interceptors

Interceptors can be added at various points. To add interceptors to a specific chain of a service, they are annotated with interface annotations or inserted programmatically. The phase at which the interceptor is inserted into the chain is determined by the interceptor itself. `Features` provide a way to add capabilities to various components in CXF. To add a feature to a service interface, the `Features` annotation can be used on the service's interface. If a CXF client uses this interface, it automatically uses the client part of the feature. `Features` can, for example, be used to add multiple interceptors, other management components, etc. An example, which uses three interceptors and a feature via annotations, can be seen in Listing 4.3.

```
@org.apache.cxf.interceptor.InInterceptors (interceptors =
    {"com.test.A","com.test.B" })
@org.apache.cxf.interceptor.OutInterceptors (interceptors = {"com.test.C"})
@org.apache.cxf.feature.Features (features =
    "qos.measuring.QoSMeasurementFeature")
public interface ITest
{
  ...
}
```

Listing 4.3: Adding Interceptors and a Feature to a Service

```
List<Header> headers = msg.getHeaders();
QName headerQName=new QName(NAMESPACE, __HEADERNAME__);
SoapHeader header = new SoapHeader(headerQName,__HEADEROBJECT__,new
    JAXBDataBinding(__HEADEROBJECTCLASS__));
header.setMustUnderstand(false);
headers.add(header);
```

Listing 4.4: Example For Adding SOAP Headers in CXF

Adding Headers

When using Apache CXF, headers can be added to a SOAP message in different ways. In this thesis, the headers are added in a SOAP interceptor, using the snippet in Listing 4.4.

4.2 Esper

This section provides a short introduction to Esper [31] and its query- and processing-language EPL. Esper is a framework for CEP. Currently, there are two versions, for Java and .NET. As the prototype in this thesis is based on Java, only the Java implementation is considered. Its documentation can be found in [39].

Contrary to relational database systems, Esper works on data streams instead of tables. This leads to an inverted scheme, where the query is fixed and the data is changing [41]. In addition to event processing, it supports using a relational database system for joining static data to data streams or other uses. As event processing is very dependent on time, Esper provides various methods of time handling, including automatic or manual time progression and isolation of different subservices [30].

4.2.1 Event Processing Concepts

Esper builds on many concepts, some already known from the Structured Query Language (SQL), some particular to CEP. The basic concept of event processing is the data stream. Based on this stream, SQL-like EPL [32] queries can filter, join and aggregate events. Additionally, it is possible to create new data streams based on queries to build complex events.

Events

An event is something which occurs once and has a timestamp and additional properties (e.g., type or name). In Esper, an event should never be modified after being submitted to the engine. It supports different kinds of event representations: Plain Old Java Objects (POJOs), XML representation, Java maps or other representations through extensions. Events also support nesting of other events.

Windows

Windows provide a data source in Esper and are used in a similar way to tables in SQL. Named windows are global windows with a name, which can be used to insert, delete or query events [32]. Mostly, they stream a specific type of event, although there are *variant streams* for which this restriction does not apply. It is possible to create a named window explicitly or to just "`insert into`" any (even previously unknown) named window and let it be created automatically. Other windows are created using views.

Views

Views [34] are a mean to constrain data streams. They can be applied to windows, patterns or other views. There are two types of views: *data window views* and *derived-value views*. *Data window views* are used to store events for a specific duration and then release them. Joins or aggregations use these windows to limit the number of used events. Typical data windows views are *time* or *length* windows, which save events for a specific duration or a specified number of events. Those two windows can be used in a tumbling manner or in the sliding default. Tumbling (or "batch") windows are evaluated and reset after a the specified time or the specified number of events have passed. Sliding windows slide over the event stream, with new events entering and old event falling out. They are reevaluated every time an event enters or leaves the window. Detailed information on windows can be found in the Esper documentation on its processing model [38].

Another important view is the *grouped data window*, which creates parallel windows for different events, based on certain grouping criteria. Other data window views include *sorted windows* or *time-order view*, both sorting the event stream. *Derived-value views* calculate new values based on the input stream. Examples are the *size* view, which calculates the number of events in a window, and statistic views (such as weighted average, univariate statistics, regression, etc.). Most views provide an insert- and a remove-stream, corresponding to the events entering and leaving the window. It is also possible to define custom views [35].

Filters

By specifying a filter on a data stream, only the events matching the filter are let through. This is an alternative to using the `WHERE`-construct (used like in SQL), but used *before* applying other constraining views (e.g., for getting the last five events with parameter A equal to 1).

```
SELECT firstEvent.id as one, followingEvent as two
FROM pattern[
  every firstEvent=AEvent ->
  (followingEvent=BEvent(service=firstEvent.service) AND NOT
      AEvent(service=firstEvent.service))
  where timer:within(2sec)]
```

Listing 4.5: Example Esper Event Pattern

Aggregations

Aggregations are used similar to SQL. They create new values based on calculations over groups or data view windows. Some predefined aggregation functions are `COUNT`, `SUM` or `AVG`. Esper also allows to define custom aggregation functions [35].

Joins

Joins provide the ability to connect two or more streams based on certain properties. Like in SQL, there are different types of joins. Joined events are generated when an event from any of the used streams arrives. An alternative is using a join, where one stream is defined as unidirectional. Only when an event comes from this stream, it is joined to the events in the other data window. Special windows can be used to access relational databases. They are used like a regular data windows, even allowing to access column values in the `WHERE`-part of the query or to join events to static SQL data.

Patterns

Patterns [33] allow to defined complex flows of events, including content-related dependencies. Some of the constructs provided are for event successions, logical expressions, sub-expression repetition or timing. The `every` keyword specifies which parts of the pattern should be repeatedly queried instead of only once. Timing constructs allow to define timers for automatic event generation, timeouts when waiting for an event succeeding another, etc. The example in Listing 4.5 shows a pattern which looks for a `BEvent` which follows an `AEvent`, arriving within two seconds and having the same `service` parameter, without another `AEvent` (with the same `service` parameter) in between.

4.2.2 Performance

According to [36], Esper provides a high performance with linear scalability between 100.000 and 500.000 events per second on a test system, using a dual CPU system with 2 Ghz. A benchmark kit allows to measure the performance on a variety of systems. A collection of measurements can be found in [37]. Some tips for improving performance can also be found in the documentation.

4.3 Frag

Frag [101] is a dynamic object-oriented programming language written in Java. It is embeddable, extensible and easily modifiable. It is possible to reduce the language to the very basics or to extend it with extra language constructs. Per default, it also includes a parser. All those points allow to use it in various approaches to DSL implementation. The language documentation, as well as a detailed description on DSL usage, can be found at [45].

4.3.1 Core Language

One of the basic components in Frag is the command. Every line of code consists of a command, possibly containing other commands. A command consists of an `object`, a `methodName` and `parameters`. Objects are a core concept of the language. They have different relationships, also including the subclass relationship. This way a complex, flexible class hierarchy can be built. Every object can be a member of different classes, it can even use itself as a class. An object has many default methods for adding classes, creating instances of itself or setting class members. Methods themselves can be added through the `method`-method, which requires a method name, a parameter list and a block of code as parameters. To define an object, the `create`-method of the object representing its class is called. The predefined objects are defined in Java, additional ones can be added via Frag.

Control Structures

All control structures are implemented as objects using parameters. Available control structures are conditionals (`if`, `switch`), loops (`while`, `for`, `foreach`) and loop control statements (`break`, `continue`). As an example, `if` can use two string parameters, the first is a condition, the second is the code block which is executed if the condition evaluates to true. The parameter strings are evaluated by the `if`-object itself, if necessary.

Mixins

Mixins are attached to objects and add functionality to them. This provides extra flexibility compared to subclassing. A mixin is essentially a collection of methods which extend or override the object's default methods. That way, method calls can be intercepted or additional methods can be added. Usages include logging, different serialization possibilities for data objects or an alternative to templates (see Section 4.3.2).

Accessing the Interpreter

Frag provides direct access to its interpreter. An `interp`-object allows access to the environment, used objects, to stop the interpretation, etc. Another important object is the `eval`-object, which executes strings as Frag code. It also allows to execute code some levels up the call stack which can be useful for adding language features.

```
Interp interpreter = new Interp();
Dual resultDual=interpreter.eval(new Dual("math add 42 9"));
NumberValue resultNumber=NumberValue.asNumber(resultDual);
int intResult=resultNumber.intValue();
```

Listing 4.6: Invocation of a Frag Command From Java

4.3.2 Using Domain-Specific Languages

In Frag, there is an extra package for DSL support [44]. It contains support for creating a language model, a configurable parser, and support for mapping the parsed tokens to the model.

Model

The model is created using classes and associations of those classes. Every model class is an instance of `FMF::Class` and can use the standard class inheritance mechanism, as every object can also be used as a class.

Associations are defined explicitly via instances of `FMF:Association`. They provide support for role names, multiplicities, navigation directions and two `FMF:AssociationEnd` objects, representing the association ends. There are two subclasses of `FMF:Association`: `FMF:Aggregation` and `FMF:Composition`. They have the same semantics as their UML counterparts. Their association ends can have an `aggregatingEnd` attribute, to specify which of both ends is the aggregating end.

Parser and Mapping

For external DSLs, Frag includes a lexical parser which uses rule definitions similar to Extended Backus-Naur Form (EBNF) [27]. It uses a custom DSL for defining these rules. A `mapping` defines, how the parsed tokens are transformed to the model. The mapping definition is specified using a mapping DSL It uses flow elements such as repetition (`rep`), sequence (`seq`) and alternative (`alt`) as well as elements (`elt`, as a representation of a token) to describe the succession of the parsed tokens. By running through this structure, it executes corresponding code, adding objects and references between objects to the language model.

Code Generation

Frag also supports code generation through the use of templates or mixins. Mixins can be used to extend the model elements with code to "serialize" them to code, which can be useful for some complex transformations. Templates are used through configurable `TemplateEngines`. They support placeholders, Frag code and direct access to the object being currently transformed.

4.3.3 Embedding into Java

Frag is written in Java and is easily embeddable into Java. To use a Frag interpreter in Java, one creates an interpreter object and uses its "eval"-method to evaluate code. Frag internally uses

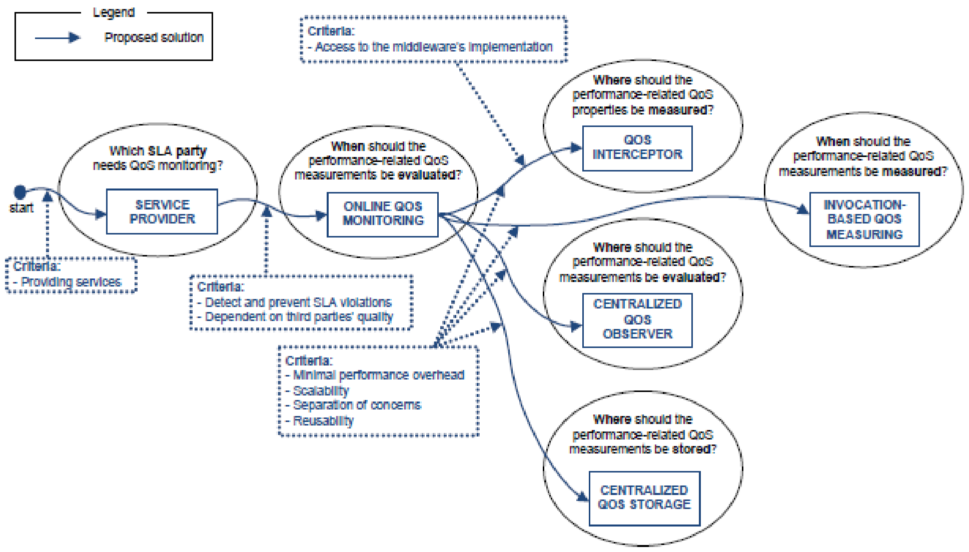

Legend
Proposed solution
Criteria:
- Access to the middleware's implementation
Which SLA party needs QoS monitoring?
SERVICE PROVIDER
start
Criteria:
- Providing services
When should the performance-related QoS measurements be evaluated?
ONLINE QOS MONITORING
Criteria:
- Detect and prevent SLA violations
- Dependent on third parties' quality
Criteria:
- Minimal performance overhead
- Scalability
- Separation of concerns
- Reusability
Where should the performance-related QoS properties be measured?
QOS INTERCEPTOR
When should the performance-related QoS measurements be measured?
INVOCATION-BASED QOS MEASURING
Where should the performance-related QoS measurements be evaluated?
CENTRALIZED QOS OBSERVER
Where should the performance-related QoS measurements be stored?
CENTRALIZED QOS STORAGE

Figure 4.2: Design Decisions for the Framework Around QuaLa(From [79])

`Dual` objects for representation for all data. Such a `Dual` represents the programming language syntax string as well as its concrete interpreted value. All methods use `Duals` as parameters and return values. For different types of values in Frag (e.g., Numbers, Lists, Objects) there are Java classes for representation and extraction from `Duals` (e.g., `NumberValue`, `ListValue`, `FragObject`). Strings can be directly cast from `Duals`. References between objects in Frag are resolved to `FragObjects`. For an example invocation, including conversion of the return type, see Listing 4.6.

To check the type of a `FragObject`, its `isType`-method can be called, but it specifically needs the type as a `FragObject`. To achieve this, the `lookupObject`-method can be used to find global objects (e.g., types) inside of Frag.

4.4 QuaLa

The QuaLa and the architecture surrounding it are described in [79]. It is a collection of DSLs to describe SLAs, services and the technical background of monitoring services. The provided architecture is used to collect measured data, organize it in a reusable way and distribute notifications of SLA violations.

The framework presented in this thesis builds on some parts of the QuaLa and extends it. It also reuses some of the architectural ideas, such as the use of a centralized monitor or interceptors for performance-based measurements.

```
# phase specifications
cxf::OutPhase create OutSetup
cxf::OutPhase create OutSetupEnding
...
## chain specifications
cxf::OutChain create ClientOut -phases {OutSetup OutSetupEnding ...}
...
## metric specification
RoundTripTime classes cxf::QoS
RoundTripTime chains ClientOut
RoundTripTime phases {OutSetup OutSetupEnding}
```

Listing 4.7: Example of the Low-Level DSL (After [79])

4.4.1 Architecture

Concerning the general architecture, [79] makes some design decisions (see Figure 4.2). The QoS data of services is measured online, meaning that it is measured as it is produced, using interceptors on each invocation. The data is sent to a centralized observer using a centralized storage.

The general workflow, using the framework, is as follows: The DSLs are transformed into a language model, which is equivalent to the domain model [79](page 100-101). Using templates, this model is converted to an Apache CXF [6] interceptor class for each measured QoS attribute. This interceptor is attached multiple times, varying its behaviour dependent on the phase (see Subsection 4.1.1) it was attached to. This interceptor then locally calculates the time difference between two phases and send them to the centralized monitor.

4.4.2 The Domain-Specific Languages

[79] presents some domain-specific languages, two of which are of relevance for this thesis. The Low-Level DSL and the High-Level DSL constitute the QuaLa. They are split to support differently skilled stakeholders. Together they describe the whole system.

The Low-Level DSL

The Low-Level DSL is used to describe the technical base of the measured services. In detail it describes three different aspects. Firstly, it describes the technical details of the measured services, similar to WSDL, plus some details for the code generation. An example is shown in Listing 4.8 Secondly, it describes the structure of the used Web service framework (Apache CXF in this case). Thirdly, it describes where and how the metrics can be measured based on the Web service framework. The latter two aspects are shown in examples in Listing 4.7.

The High-Level DSL

This DSL is used for the description of SLAs (see Listing 4.9). A SLA has a name and can be used for many services, which are referenced using a name. For each of those services, multiple

```
## LOGIN SERVICE
ExampleService classes cxf::Service
ExampleService package "at.happens.exampleservice"
ExampleService uri "http://localhost:5001/services/exampleservice"
ExampleService wsdl "http://localhost:5001/services/exampleservice?wsdl"
ExampleService namespace "http://happens.at/services/exampleservice"
ExampleService operations [list build \
  [cxf::Operation create echo -name "echo" -returnType String -parameters
      [list build \
      [cxf::Parameter create echoParam -name "echoParameter" -type String]
      [cxf::Parameter create auth -name "authentication" -type String]
    ]
  ]
]
```

Listing 4.8: Example of the Low-Level DSL, Specifying a Service (After [79])

```
TheSLA {
  ServiceA {
    UpTime>99% AND ProcessingTime<1min => smsto "+00 000 000 000",
    DeliveryRate>70% => mailto "x@x.com"
  }
  ServiceB {
    ProcessingTime<2min => smsto "+00 000 000 000"
  }
}
```

Listing 4.9: Example of the High-Level DSL

rules can be defined. Every rule consists of a condition (equivalent to a SLOs) and an action. The action describes who should be notified and how if the condition is violated. The QoS attributes used in the conditions are predefined in the syntax and the language model.

4.4.3 Supported Measurements

The framework supports the measuring performance-related metrics (see Figure 4.3). It distinguishes between *negotiable*, *network-specific* and *provider-relevant* performance-related QoS properties. Negotiable properties are properties which are usually negotiated in SLAs. Examples are round trip time, processing time, response time or up-time. Network-specific properties are usually not mentioned in an SLA but can be used to detect bottlenecks in service invocations. They include marshaling time, execution time or network latency. Provider-relevant properties describe the performance of the service provider, such as throughput, scalability and robustness. The framework uses dedicated interceptors for each supported metric.

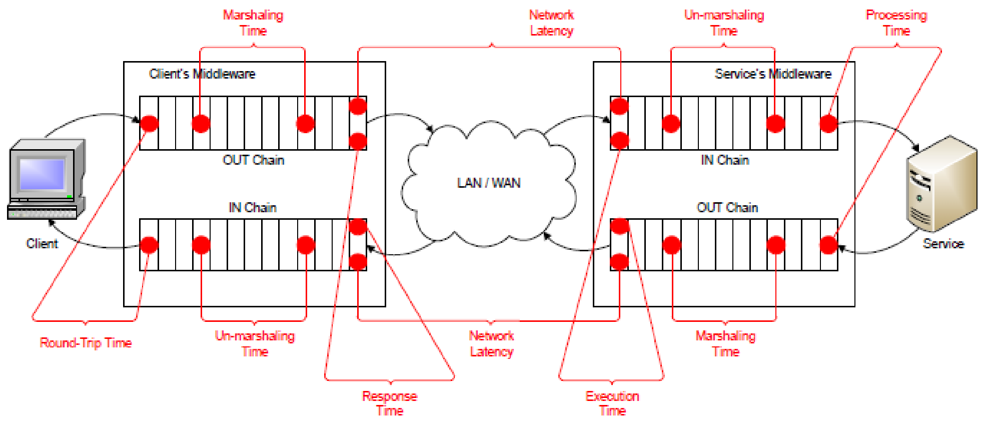


Figure 4.3: Measuring Points for Some Performance-Related Metrics (From [79])

5. Design

This section presents a motivating example, followed by a basic description of our solution. The architecture of the different components and their interaction is shown in the next subsection. Finally, the DSLs used for configuring and controlling the system are described in detail. While discussing the solution, the differences and similarities to the QuaLa approach (see Section 4.4) are highlighted.

5.1 Motivating Example

As a base for this chapter, this section presents a motivating example of a monitored Web service. The example service is used to upload, search and organize music online. It can be used to store your music database online and access it from everywhere. In addition, it proposes new songs or Internet radios based on currently owned songs.

It implements the following interfaces:

Interface	**Methods**	**Explanation**
ILogin	`boolean login(String user,String passwd)`	Log into the service.
	`void logout()`	Log out of the service.
IManage	`void upload()`	Load a music file onto the platform.
	`void rename(int id,String newName)`	Rename an uploaded file.
	`void delete(int id)`	Delete an uploaded file.
	`Metadata[] list()`	List all uploaded files.
	`MusicFile get(int id)`	Get a specific file by ID.
IQuery	`Metadata[] simpleSearch(String text)`	Search all metadata for a simple search string.
	`URL[] proposeRadioStations(int count)`	Let the service propose a radio station, based on the currently uploaded music.

Using such a Web Service involves different stakeholders with different needs. To agree on the guaranteed QoS of this service, constraints are defined in an SLA. Clients choose the service they are going to use based on the guaranteed QoS from the SLA and the measured QoS.

If a QoS guarantee is violated, the clients want compensations or penalties for the service operator. They also want to be able to rate certain service qualities. The service operator wants to ensure its conformance to the SLA. He tries to adapt its infrastructure and processing mecha-

	Attribute	**measured at**		**Explanation**
		client	**service**	
Performance-related	Processing Time	☐	☑	Time it takes for the server to execute a called method (excluding marshalling, etc.)
	Round Trip Time	☑	☐	Overall time it takes for a remote call to return control.
	Error Rate	☐	☑	Percentage of exceptions in a certain time span.
	Load	☐	☑	Number of invocations in a certain time span.
	Network Latency	☑	☑	The time it takes invocation and response to travel over the network.
	Latency	☑	☑	The time invocation and response travel through the network and the framework.
Availability-related	Server Availability	☐	☑	Percentage of time the server is online, calculated via heartbeats.
	Client Availability	☑	☐	Percentage of time the server is online, calculated via reachability from clients.
	Mean Time Betw. Failures (MTBF)	☑	☐	Mean time between two separate failures of a service.
	Mean Time To Failure (MTTF)	☑	☐	Mean time it takes from a service going online to it failing.
	Mean Time To Repair (MTTR)	☑	☐	Mean time it takes from a service failure to the repair.
App.-spec.	Search Result Accuracy	☑	☑	Percentage of successful finds in all search requests.
	Proposition Quality	☑	☐	Subjective quality of a proposition, rated by the client.

Table 5.1: The Measured QoS Attributes and Their Measurement Location

nisms to prevent penalties, arising from contract violations. In addition it wants to keep some user perceived QoS attributes (e.g., round trip time) low, to be an interesting choice for service selection. To achieve this, the system uses our approach to QoS monitoring.

The currently implemented QoS attributes and where they are measured is shown in Table 5.1.

5.2 Architecture

To overcome this challenge, this thesis proposes a system for QoS monitoring using DSLs and CEP. The system is comprised of services, clients and a separate central monitor. It supports the definition of SLAs, violation checking through flexible QoS measurement and actions in case of SLA violations. For definition of the SLAs, a modified version of the *High-Level QuaLa* from [79] (see Section 4.4.2) is used. It supports condition-action rules, with conditions being `boolean` expressions on metrics, defined in the MDDSL. Supported actions can be added using plugins.

The basic architecture adds a monitoring component to the SOA triangle. As the registry is not relevant for this approach, only the three main components, and the `Web interface` are shown in Figure 5.2.

The three components are:

- `Service`: The monitored service, providing functionality. Houses a library to support measurement by the framework.
- `Client`: The client, using the service. Can use the library to support additional client-side measurements.
- `Monitor`: Used for measurement administration as well as the collection and evaluation of measurements.

The same library can be used for clients and services. For some systems which are both, synergy effects (e.g., reducing some sending overhead) can be taken advantage of.

Flexible QoS measuring is implemented by using the MDDSL to describe how to monitor a specific metric. Those metrics are used in the SLA, where constraints on them are formulated for specific services and actions are defined, based on existing action plugins. To describe a metric, Measurement Descriptions (MDs) are used, formulated in the MDDSL. It uses `functions` to build metrics from basic events, originating from Measuring Points (MPs). MPs are predefined, parametrized data sources on clients or services. An example MP could attach itself to a phase in an Apache CXF (see Section 4.1) invocation chain and fire events containing information about the corresponding invocation. The SLA, the measured services and the MDs are stored and managed only in the central monitor. At runtime, the needed MPs are calculated from the SLA and, the information is distributed to the corresponding services through the Measuring Location Framework (MLF), which is the part of the framework residing at the clients or services. When an MP fires, the framework sends the data to the central monitor, which uses the MD to calculate the metrics. The monitor translates the MDDSL to event transformations. If it receives a measurement event, it uses them to calculate the metrics. The constraints and action specification in the SLA are also translated to transformation rules, to allow the CEP system to check for SLA violations. A depiction of the process is shown in Figure 5.1. In case of a violation, an `ActionEvent` is sent. The `Action System` checks for such events, and triggers

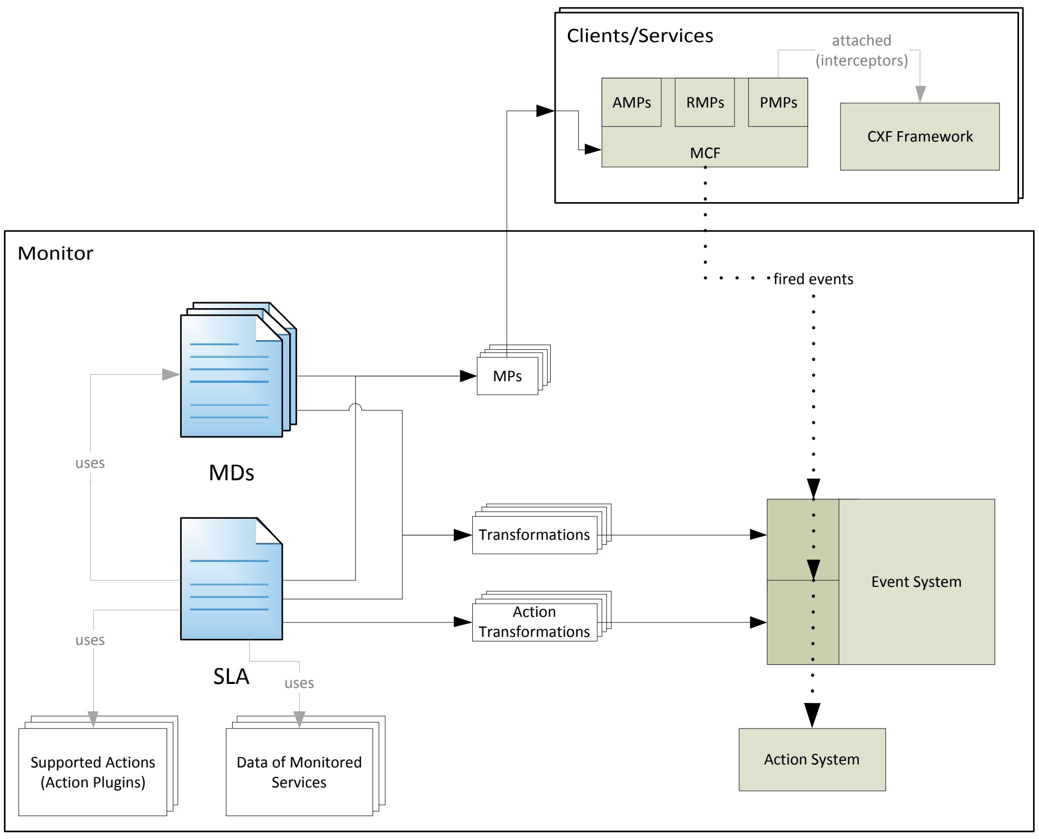


Figure 5.1: Data Flow of the DSL Documents

the corresponding actions. All events and transformations support causality, enabling a notified component to detect the cause of an SLA violation.

The process can also be seen as two separate circuits (see Figure 5.2):

- *MD distribution* (blue): Via the `Web interface`, the stakeholders can edit their artifacts in the monitor. Monitored services can be registered and new MDs can be defined. Both can be used in the SLA which spans various services. The monitor distributes the information which data has to be measured (the MPs) to the clients and/or services. Clients poll this information if needed, as there is often no way to determine or address the clients before the invocation. Registered services are notified actively through their `Distribution Interface`.
- *Measuring* (black): MPs are attached to the services or clients. If a MPs has data to send (for example, if a certain step of an invocation is reached), the framework sends its measured data to the monitor as an event. This events also contains the name of the MP for

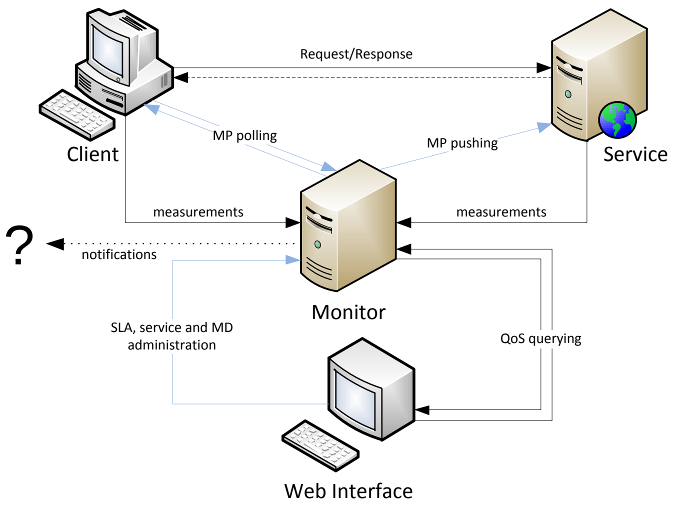


Figure 5.2: Overview of the System Architecture

identification reasons. On the monitor, the CEP transformations convert the simple events to metrics through a series of intermediate complex events, maintaining causal relations. The monitor stores this metric data and verifies the SLA. If a violation is detected, its `Action System` fires an action and notifies the agent (stakeholder, computer program, etc.) specified in the SLA. The QoS data can also be queried through a Web interface.

5.2.1 Measuring Location Framework

The concrete measurement takes place in a framework at the services and clients location, called the MLF. A comparison of different measurement locations can be found in [79]. It found that a proxy-based approach has a big performance overhead, while an interceptor-based is comparable to an inline approach. Another possible approach would be aspect-oriented weaving of code into the service, but the interceptor approach provides additional flexibility. Specifically, it allows to intercept method calls at various phases (see Subsection 4.1.1). This allows to measure steps of the calling process, such as marshalling or the raw network sending time. In addition to this measurement, other more general measurements can also be taken if the corresponding MPs are supported. Although the raw data (mostly timestamps and call metadata) is measured on site, the data is sent to the central monitor for processing. This avoids the overhead of calculations at the service and therefore reduces measurement distortion. It also allows additional validity checks at the monitor.

The MLF has two important functions: retrieving of MP updates and measuring of QoS data. They are implemented using three basic components: A `Distribution Interface`, the `MP Management Subsystem` and the `Sending Subsystem`. The `Distribution Interface` exists only for services who get notified of MP changes by the monitor, but not for clients, as they instead poll the MP updates of the used services. It only has one method, which accepts a service Uniform Resource Locator (URL), a timestamp and a list of MPs which should be measured. The `MP Management Subsystem` is separated into one QoS manager for each used or provided service. It stores the currently used MPs and attaches them at the appropriate position. If an MP is no more used, it is removed from its location and deleted.

There are many different possible types of MPs. Three basic types, important for the current task are:

- *Application-Specific Measuring Points (AMPs)*: MPs fired by the application or the client for application-specific measurements. Even though they are not measured automatically, they are registered to allow the MLF to discard invalid measurements. There are two types of AMPs: Application-Specific Time Measuring Points (AtMPs) and Application-Specific Value Measuring Points (AvMPs), which can be used to send values or timestamps, respectively.
- *Repeated Measuring Points (RMPs)*: MPs which fire repeatedly in a certain interval. This is used for some availability-related measurements
- *Phase Measuring Points (PMPs)*: MPs which are attached to a phase in a method call. To allow a very fine placement, PMPs are technology-dependent. Other Web service frameworks have different phases or no interceptor system at all. In this case, other MPs take over this place.

There are lots of other possible MPs which could extend the functionality of the system. Some are mentioned in Section 7.2.3.

If an attached MP is activated, an event is sent to the `Sending Subsystem`. This event contains the name of the MP which uniquely identifies is and IDs for the current call and method. This name is created from the service address, the type and parameters of the MP (see Section 5.3.3). The `call-` and `method-IDs` allow correlation of a MP to the method of an interface or to a specific call. While the `method-ID` can be calculated by the interface, the `call-ID` is generated by the MLF on the client for each invocation and sent to the service with the message. The `Sending Subsystem` is used instead of directly sending the request for two main reasons. First, it caches measurements and sends them grouped, reducing the sending overhead. Second, it is able to order events based on their arrival, keeping the timestamp order intact for each service. Therefore the monitor does not have to handle out-of-order events and the MDs can be kept simple. To avoid delays, because the events are sent to the monitor, they are queued and send by another thread.

5.2.2 Monitor

The monitor acts as a separate management- and measurement-instance. Calculating of the metrics is separated from the measurement location to reduce the calculation impact on site. The separation also allows some independent measurements which are inherently more reliable than measurements reported from a foreign computer system. Measured QoS information is centrally stored at the monitor, but it could also be sent to a registry.

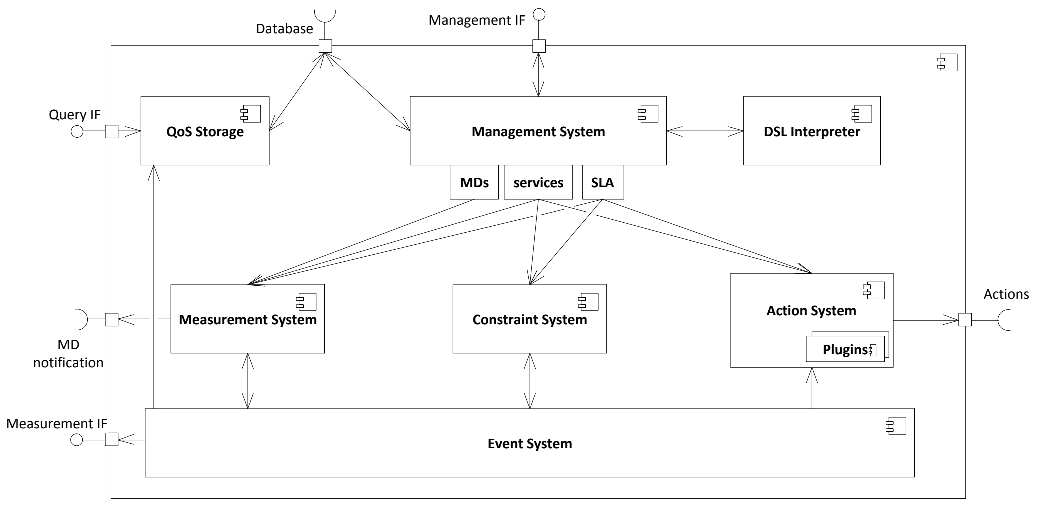


Figure 5.3: Components of the Monitor

In Figure 5.3, the general structure of the monitor is shown. Three Interfaces provide the basic communication with the outside. The `Management Interface` is used for management of SLAs, MDs and services. The `Measurement Interface` receives measurements from the MLF. The `Query Interface` can be used to query current QoS values and historic data.

Updates through the `Management Interface` are sent directly to the `Management System`. It stores the SLAs, MDs and services into the database and translates the SLAs and MDs to CEP transformations (see Section 5.3.3). When transforming an SLA, it checks for the used metrics and collects the needed MPs from their corresponding MDs. The event transformations gained from translating the SLA and MDs are stored into the `Measurement System`. Accompanying the transformations, the needed MPs (and the services they are needed from) are also submitted to the `Measurement System`, which notifies the matching services.

In the `Measurement System`, the simple events, received from the `Measurement Interface`, are transformed in various steps to complex QoS metrics, based on the transformations gained from the MDs. To detect violations of the SLA, the SLOs and the corresponding actions of the SLA are also translated into event transformations. They are stored into the `Constraint System` which waits for changes in the QoS metric values and fires action events in case of a constraint violation. These events are collected by the `Action System`. The `Action System` has a plugin feature to support different action types, such as HTTP invocations, EMail or simple console output. Other types of plugins could easily be attached. The plugins are activated if a corresponding event was fired, receiving the action event, including the complete causal history of the constraint violation. A possible adaptation component, notified through a plugin, could then adapt the service according to the cause of the SLA violation.

5.3 DSLs

There are three main DSLs used in this approach. The *MPDSL* describes a model of the infrastructure of the measured system for use in MP definition. It based on the Low-Level DSL from [79] but heavily reduced and extended to define MPs. The *Measurement Description DSL (MDDSL)* allows to define QoS metrics (so-called MDs), based on those defined MPs. They are used by the *SLADSL* to describe SLAs. The SLADSL supports different services, constraints (SLOs) and actions to be taken if a constraint is violated or its violation ceases. While it is based on the High-Level DSL from [79], some minor syntactical additions are made. The main changes lie in the processing of the DSL.

5.3.1 MPDSL

The MPDSL is an internal DSL, using Frag (see Section 4.3) as its host language. Its syntax is used from Frag. Its QuaLa pendant was a part of the Low-Level DSL, used to describe the technical structure of the framework of the measured system and for code generation (see Section 4.4.2. As the client code is no longer generated from the DSLs, but the MLF only interprets the DSLs, the service model (including operations and parameters) was removed. The services are now stored separately, to allow exchanging at runtime. The SLA only references them through their assigned names.

The QuaLa defined the metrics only by specifying a list of phases, where the interceptors should be attached. In the refinement, metric definition is moved to the new MDDSL. The new MPDSL is used to describe the model for a specific Web service framework (e.g., Apache CXF). It also defines the supported MPs of this framework. Those MPs can then be used in the MDDSL.

To support a new framework in the DSL, including MPs, at least one of two `Measuring-Point` classes of the MDDSL model have to be extended. The `TimeMeasuringPoint` is for MPs which send timestamp data to the monitor. The `ValueMeasuringPoint` is for MPs which send a numerical value to the monitor. Those two classes are part of the MDDSL model. Other classes can be introduced to model the service framework and parametrize the MPs. In addition to extending one of the two classes, every MP has to have a `getMPdata` function, which receives a service name and returns a Frag object. This object should define a globally unique MP type, a location (CLIENT/SERVER), other attributes with string values for all relevant properties, and a unique MP name, which contains type, location and all relevant properties which make this MP unique. The `DSLResolver` converts it to a Java `MeasuringPoint` object for distribution to the MLFs. Definitions of the used MPs can be found in Section B.3.

As the MLF needs to know how to measure a certain MP, defining it here is not enough, it also has to have an implementation at the MLF. By examining its parameters, the MLF is able to find a place to "attach" it and to determine parameters some other features of the MP (e.g., time intervals). When the MLF fires an MP, it creates a `QoSTimeEvent` or a `QoSValueEvent`, depending on the type of MP. This event contains the measured value, the MP's name, location (`CLIENT/SERVER`), service address and potentially a call- and a method-ID. The model of the CXF framework that was used in the prototype, can be seen in Figure 5.4.

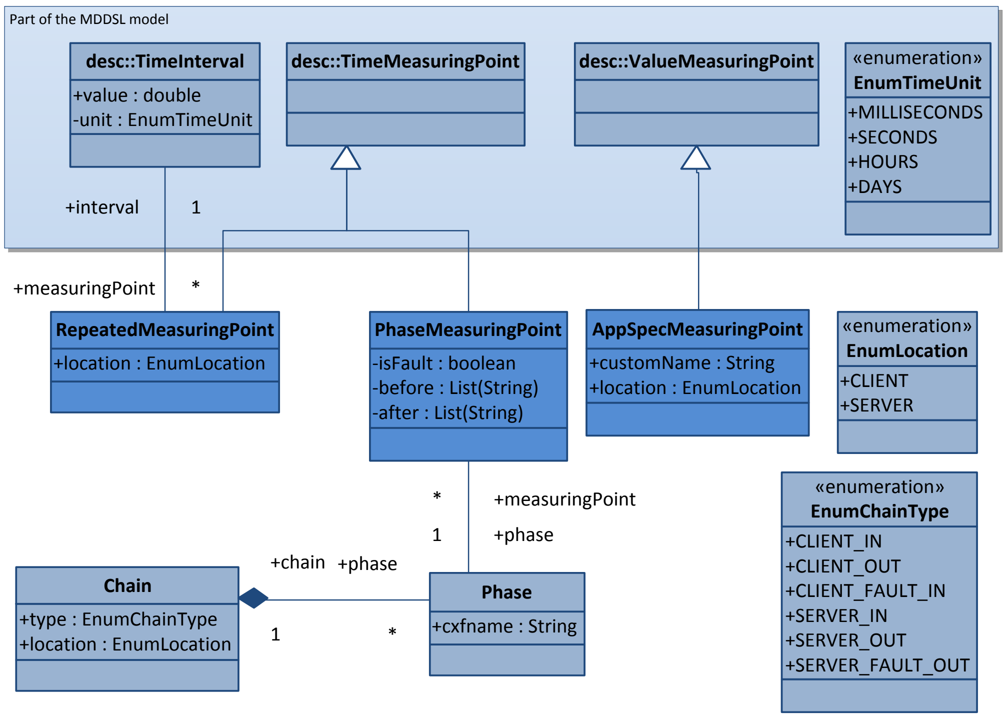


Figure 5.4: Example Model for the CXF Framework, Originally Defined in the MPDSL

5.3.2 SLADSL

The SLADSL is an external DSL, based on the High-Level DSL of the QuaLa. Model and syntax were adapted to fit runtime processing instead of code generation and to allow to filter unwantedly repeated events. It is used to define SLAs using constraints (corresponding to SLOs) and actions which are evoked if the constraints are violated.

Model

The structure of the model for the DSL is shown in Figure 5.5. Every SLA defined by the SLADSL has a name and can span a number of services, referenced by their name. For each service, `constraints` are defined. A `constraint` consists of a repetition type, a `condition` and an `action`. The `repetition type` defines when an action event should be fired:

- `WHEN_NOT`: The event is fired when the condition is first violated (evaluates to false). If the `condition` later evaluates to true and then to false again, it fires again.
- `WHILE_NOT`: The event is fired every time the value of one of the metrics changes, but only while the `condition` is violated.

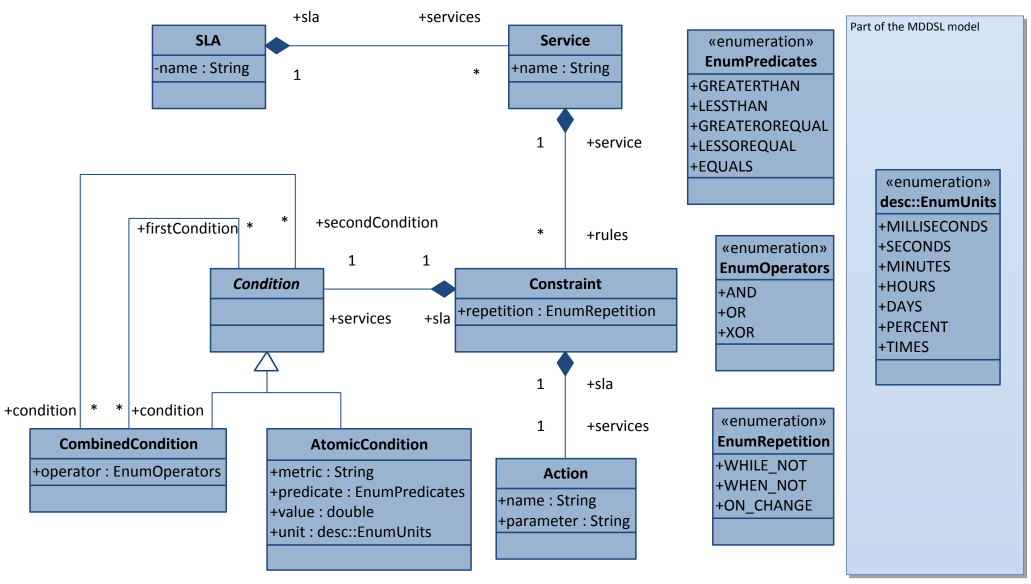


Figure 5.5: Model of the SLADSL

- ON_CHANGE: The event fires when the violation status changes from false to true or backwards. Intermediate events are dropped.

This way, superfluous action events can be prevented.

The condition is a boolean expression. An atomic condition consists of a comparison between a metric name and a double value. A combined condition uses two conditions and a boolean operator (AND/OR) to build complex expressions. The metric names must reference the name of an MD, defined in the MDDSL.

An action consist of the name of an action and a parameter. While the QuaLa High-Level DSL uses fixed names for the actions, the SLADSL aims to be more flexible by referencing the plugin names of the Action System. This adds the need to check for the existence of the necessary plugins, when the SLA has been edited. If an action is invoked, the parameter as well as the events causing the invocations are sent to the corresponding plugin.

Syntax

The syntax of the SLADSL corresponds closely to the model. One thing to be noted is, that multiple combined conditions do not have to be put in brackets. They are then evaluated from left to right. Also short-circuit evaluation is not implemented. An example of the syntax can be seen in Listing 5.1.

```
ExampleSLA
{
  Login
  {
    WHEN NOT Reputation>95% => console "err",
    ON CHANGE ServerAvailability>99% => mailto "matthias@happens.at",
    WHILE NOT ClientAvailability>95% => console "out",
    WHILE NOT (Latency>1ms OR NetworkLatency>1ms) AND RoundTripTime<100ms =>
        http "http://localhost:80/script",
    WHEN NOT ProcessingTime>30ms => mailto "matthias@happens.at",
    WHILE NOT Load<8times AND ErrorRate<95% => console "out",
    ON CHANGE MeanTimeBetweenFailures>2ms AND (MeanTimeToRepair>3ms OR
        MeanTimeToFailure>4ms) => console "err"
  }
}
```

Listing 5.1: Example SLA, Written in the SLADSL

Transformation to EPL

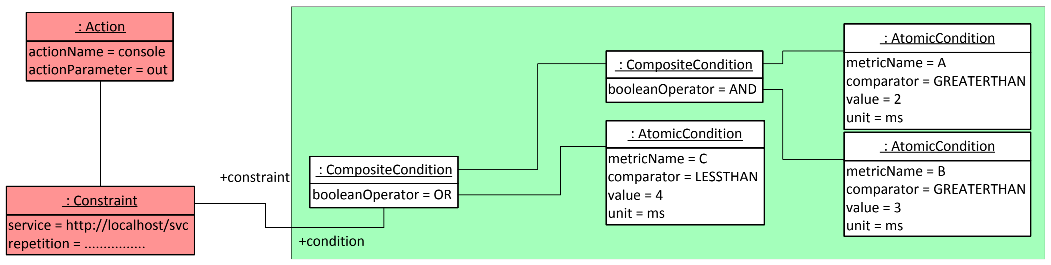
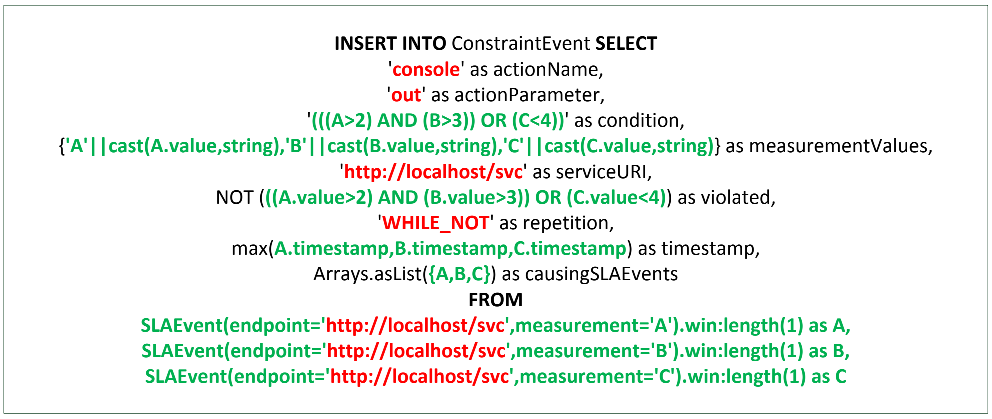


Figure 5.6: Example Object Model of the SLADSL

The predecessor of this DSL (the High-Level QuaLa) was used to build a skeleton for a service. It therefore used the additional information from the Low-Level QuaLa to create the necessary classes, interceptors, etc. Our approach uses the new SLADSL at runtime. It is not converted to Java code, instead EPL (see Subsection 4.2.1) transformations are generated. This way the values, which are measured for each metric, are transformed in a series of steps to action events, which can be picked up by the plugins. Figure 5.6 shows a part of an object model as it would be created from an SLADSL specification. It is used as a base for the transformations below.

When parsing the SLADSL, two transformations are generated for each `constraint` (see Figure 5.7). The first transformation is used to calculate if the `condition` is violated or not. It is evaluated every time a measurement changes. Every time a measurement changes, it creates a `ConstraintEvents` from the required measurements. `actionName` and `actionParameter` are retrieved from the `Action` object. `repetition`, `serviceURI` and the `endpoints` in the `FROM`-clause are retrieved from the `Constraint` object.



```
INSERT INTO ConstraintEvent SELECT
'console' as actionName,
'out' as actionParameter,
'(((A>2) AND (B>3)) OR (C<4))' as condition,
{'A'||cast(A.value,string),'B'||cast(B.value,string),'C'||cast(C.value,string)} as measurementValues,
'http://localhost/svc' as serviceURI,
NOT (((A.value>2) AND (B.value>3)) OR (C.value<4)) as violated,
'WHILE_NOT' as repetition,
max(A.timestamp,B.timestamp,C.timestamp) as timestamp,
Arrays.asList({A,B,C}) as causingSLAEvents
FROM
SLAEvent(endpoint='http://localhost/svc',measurement='A').win:length(1) as A,
SLAEvent(endpoint='http://localhost/svc',measurement='B').win:length(1) as B,
SLAEvent(endpoint='http://localhost/svc',measurement='C').win:length(1) as C
```

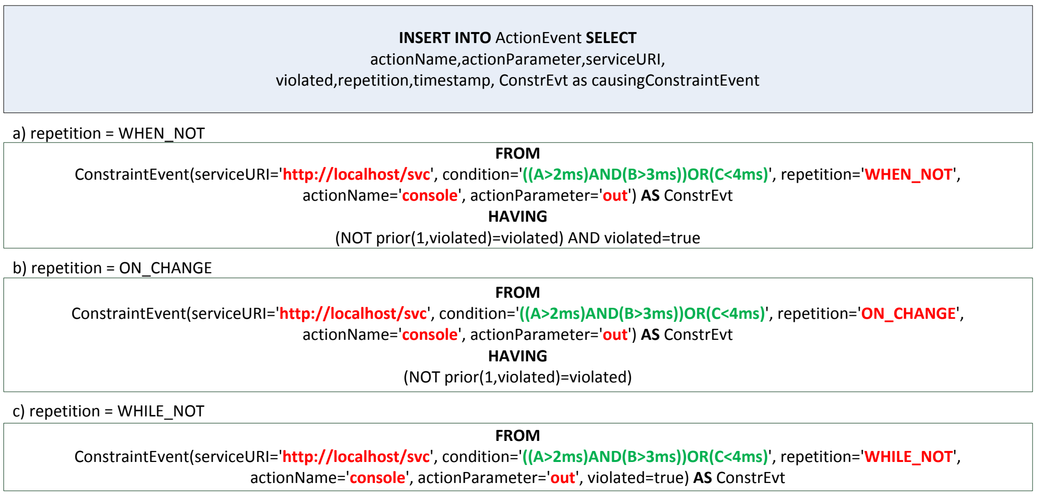
Figure 5.7: Transformation From Measurements (SLAEvents) to Constraint Events

The other (green) parameters are retrieved from the `condition` tree:

- `condition` is a reformatted `condition` string for later evaluation by the action plugin.
- `measurementValues` contains pairs of metrics and their values.
- `violated` calculates if the `condition` was violated or not.
- `timestamp` takes the timestamp of the last received event and makes it the `ConstraintEvents` timestamp.
- `causingSLAEvents` contains references to the metric events that caused this `ConstraintEvent`.

The `FROM`-clause opens a stream for each metric, joining them, while retaining only the last measurement for a metric. This causes a `ConstraintEvent` to fire, every time a one of the measurements changes.

The output events of the condition transformations are the base for the repetition transformations. Depending on the repetition type, one of three transformations is chosen (see Figure 5.8). The first part of the transformations is always the same, copying relevant values from the `ConstraintEvent` and adding the `ConstraintEvent` as the `causingConstraintEvent`. The second part (a, b or c) is determined by the `repetition type`. While the `serviceURI` can be fetched from the `constraint`, the `condition` is generated from the `condition` tree. Through referencing previous values of a property (`violated`), these transformations filter the unneeded events from the stream and create `ActionEvents` only for "interesting" events.



```
INSERT INTO ActionEvent SELECT
actionName,actionParameter,serviceURI,
violated,repetition,timestamp, ConstrEvt as causingConstraintEvent
```

a) repetition = WHEN_NOT

```
FROM
ConstraintEvent(serviceURI='http://localhost/svc', condition='((A>2ms)AND(B>3ms))OR(C<4ms)', repetition='WHEN_NOT',
actionName='console', actionParameter='out') AS ConstrEvt
HAVING
(NOT prior(1,violated)=violated) AND violated=true
```

b) repetition = ON_CHANGE

```
FROM
ConstraintEvent(serviceURI='http://localhost/svc', condition='((A>2ms)AND(B>3ms))OR(C<4ms)', repetition='ON_CHANGE',
actionName='console', actionParameter='out') AS ConstrEvt
HAVING
(NOT prior(1,violated)=violated)
```

c) repetition = WHILE_NOT

```
FROM
ConstraintEvent(serviceURI='http://localhost/svc', condition='((A>2ms)AND(B>3ms))OR(C<4ms)', repetition='WHILE_NOT',
actionName='console', actionParameter='out', violated=true) AS ConstrEvt
```

Figure 5.8: Transformation From Constraint Events to Action Events

5.3.3 MDDSL

The MDDSL is an internal DSL, used for describing how QoS measurements should be taken. It uses Frag As a host language, taking advantage of the programming language constructs, especially lists, references and potentially loops or calculations. The measurements are described in the form of MDs, where each MD describes one way to measure a metric. It works by utilizing MPs, defined in the MPDSL, and applying various functions to calculate complex metrics. It also adds a unit- and a name-parameter to the metric.

Based on the QuaLa Low-Level DSL, where metrics are defined at design time of the service, the MDDSL goes a step further and allows to define more complex measurements, which additionally can be changed at runtime. It allows to define custom metrics as well as detailed descriptions of how to measure performance-based, availability-based or other metrics. The predefined functions are transformed into EPL transformations, following a CEP approach. The EPL brings a huge flexibility and also a lot of complexity because of the variety of possible metrics. The MDDSL strives to simplify the metric creation process, hiding the complexity of EPL as well as the whole structure of the event system. This allows the user to focus on the metrics instead of the technical details and potential traps of event processing.

Model

As the MDDSL uses an internal syntax, it consists only of Frag constructs. The model is shown in Figure 5.9. To create am MD, a `Metric` class and function classes of the MDDSL model and MPs from the MPDSL have to be instantiated. The main object for an MD is the `Metric` object, which defines the `unit`, the `name` and a function how to calculate the metric. At the moment, valid units are: `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, `DAYS`, `PERCENT`

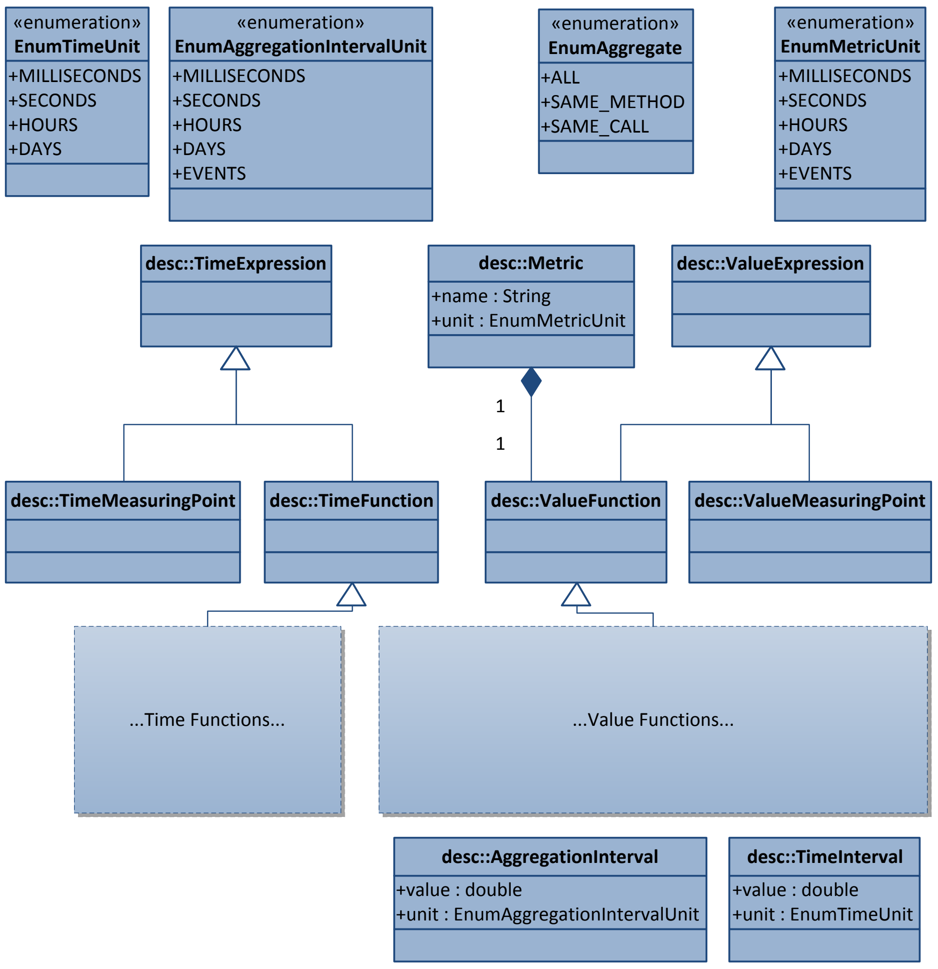


Figure 5.9: Model of the MDDSL (With Placeholder for the Functions)

and `TIMES`. In the SLADSL they correspond to `ms`, `sec`, `min`, `h`, `d`, `%` and `times`, although `times` can be left out, as it is the default. The `function` parameter requires a function, which produces a `QoSValueEvent`.

The main part of the MDDSL are the predefined functions. By implementing various needed constructs, they move the abstraction from the very technical event-based processing language to a higher level. Every function has parameters. They can be any kind of Frag object or any `TimeExpression` or `ValueExpression`. This includes MPs or other functions. By nesting functions, even complex time-based metrics can be defined. Some of the functions are aggregations. They aggregate events over time or over a certain number of events and create a new event based on the collected events. These functions have two different aggregation types:

SAME_CALL or SAME_METHOD. When stating the aggregation attribute, the function calculates the new event separately for different calls or different methods. This is based on the call-ID or method-ID which is sent with some MPs. It allows to define, for example, the time between two specific events in a single call or more complex things, such as calculating the overall average of the cumulative (*sum*) processing time of a single method in a certain interval. If the aggregation attribute is left empty, all events are taken into account. Aggregation functions also use aggregation intervals. They require a timespan or a number of events to be specified through an AggregationInterval object.

An example visualization for a metric can be seen in Figure 5.10. Other metrics, expressed in the MDDSL, can be found in Section B.1.

Predefined Functions

The predefined functions can be separated into functions which create TimeEvents (called TimeFunctions) and functions which create ValueEvents (called ValueFunctions).

The MDDSL defines twelve ValueFunctions:

Function	**Parameters**	**Description**

SUM	of(valueevent)
	interval(interval)
	aggregation(aggType)
		The sum of all ValueEvents in a certain interval.

AVG	of(valueevent)
	interval(interval)
	aggregation(aggType)
		The average value of all ValueEvents in a certain interval.

IF	value(valueevent)
	comparison(string)
	then(double),
	else(double)
		comparison consists of a comparison operator and a numerical value. The value is compared against this value using the operator. If the comparison evaluates to true, the resulting ValueEvent has the value defined in then or else the value of else.

INTERVAL	between(timeevent)
	aggregation(aggType)
		Calculates the time between every two occurences of the specified event.

LIMIT	of(valueevent)
	lower(double)
	upper(double)
	lowerInclusive(boolean)
	upperInclusive(boolean)
		Only forwards the ValueEvents with values between the upper and lower bound. Allows to specify if the upper and lower bounds are inclusive or exclusive

CALCULATION	calculation(string)
	parameters(hashtable)
	aggregation(aggType)
		Uses a hashtable to fill the events into the calculation, according to their name. Creates a ValueEvent with the resulting value.

VALUEUNION	events(valueevents)	Joins multiple streams of ValueEvents into one.

TIMEBETWEEN	from(timeevent)
	to(timeevent)
	aggregation(aggType)
		Calculates the elapsed time between the two `TimeEvents`, based on their timestamp.

MONITORTIMEAT	at(timeevent)	Calculates the time at the monitor and sets it as a value, when a certain event arrives.

COUNT	of(timeevent)
	interval(interval)
	aggregation(aggType)
		The number of `TimeEvents` in a certain interval.

TUMBLING_COUNT	of(timeevent)
	interval(interval)
		Same as COUNT, but resets after the specified timespan instead of letting old events fall out.

SETVALUEONEVENT	on(timeevent)
	value(double)
		Creates a `ValueEvent` with the corresponding value, when an certain event is received.

Table 5.2: The MDDSL `ValueFunctions`

The other two defined functions are `TimeFunctions`:

Function	**Parameters**	**Description**
ONEVENTCHANGE	events (timeevents) fireOn(timeevents)	Listens to a stream consisting of all specified `events`. If the stream changes from one event type to another which was specified in `fireOn`, it forwards the event.
UNION	events(timeevents)	Joins multiple streams of `TimeEvents` into one.

Table 5.3: The MDDSL `TimeFunctions`

As the time on the monitor is not necessarily equal to the time on the client and/or service (see also Section 5.3.3), all `TimeEvents` use the time at their source, except when coming from a `TimeFunction`. `TimeFunctions` reuse the timestamps of the client/service.

An example of how to use PMPs and the predefined functions to describe the network latency (see Section 5.1) can be seen in Figure 5.10. All metrics which are used in the motivating example, can be found as MDDSL scripts in Section B.1.

Challenges

When coordinating events from three different sources (client, service and monitor), time plays an important role. It can not be assumed that every source has the same time, especially when trying to measure at millisecond level. Because of this problem, it is not possible to directly

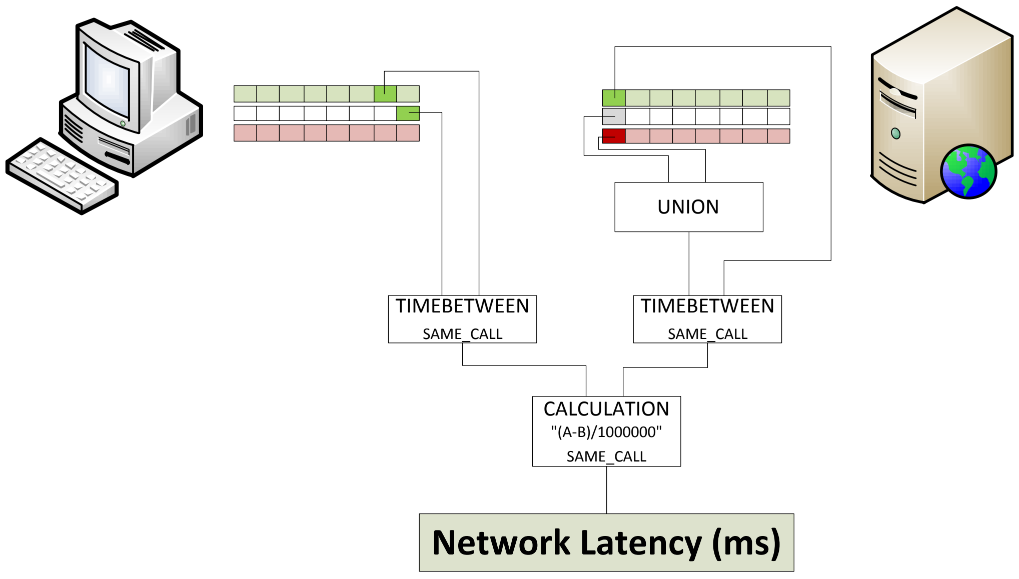


Figure 5.10: Visualization of the Network Latency - Measurement Description

measure, for example, the time between the sending on the client and the receiving at the service. For this thesis the following methods were considered as solutions for this problem :

- *Clock synchronization*: When synchronizing the clocks, using a clock synchronization protocol (e.g., the Network Time Protocol (NTP) [78]) typical accuracy is very heterogenous. Local networks generally have a much higher accuracy than networks with higher load and worse predictability, for example, the Internet. Nevertheless, some papers use some kind of clock synchronization (e.g., [96]).
- *Using monitor time*: By using the event arrival time at the monitor, only a single time source would be needed. This would introduce the need to send the events instantly, as they are measured, and add a big overhead to the measurement, because of the sending process and the networking delay between the points of measurement and the monitor. It would also skew the results when comparing sources, which have a different network distance to the monitor.
- *Staying time difference independent*: This could be achieved by designing the metrics in a way which does not calculate time differences between different time sources. For the network latency, this means that metric designers should not measure the time between the client sending and the service receiving, but the round trip time and subtract the time between the receiving and the sending of the response at the service. This requires some additional minor knowledge of distributed systems.

Our solution uses the third approach, dealing with the time differences at metric level. Yet, the first approach is not actively prevented. If the clock is synchronized externally, calculations could be made between different time zones. Using the third approach adds some requirements to the metric generation. As the monitor has a different timezone than the measuring points, aggregations can not be implemented accurately when not using monitor time. Although the CEP system has the possibility to use object timestamps to advance time instead of the system time, a sliding aggregator (e.g., `SUM`) should also update when an object leaves the relevant time window. This is not possible if the time of the measuring point is not permanently updated. Therefore all aggregation functions use monitor time.

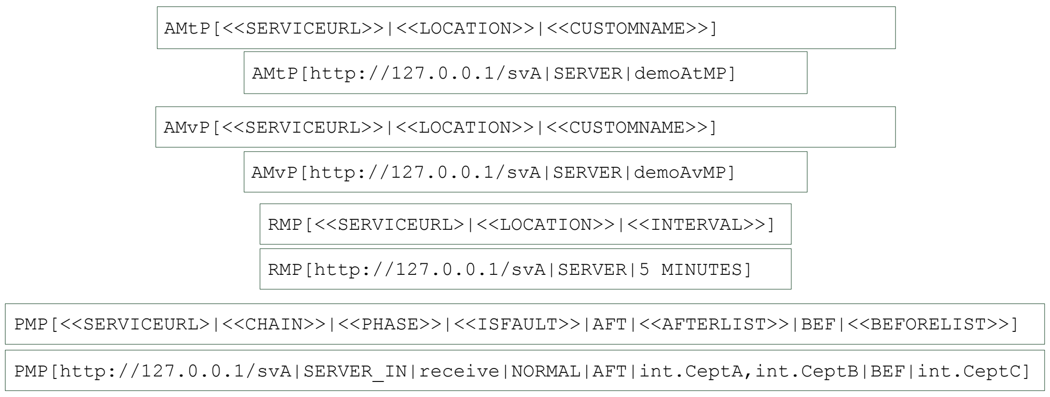
Another challenge is the ordering of events. If events are just sent as the occur, there is no guarantee that they arrive at the monitor in order, because of the network delay. An option would be restoring the order at the monitor, introducing some weaknesses, such as a potential notification delay when an event goes missing or an additional calculation overhead. Another option would be to ignore the ordering at the DSL level. This would make the language more complex and the metrics more difficult to understand and to create. Our solution uses the fact that the MLF already caches the events before sending for performance reasons. It is possible to ensure the order of the events when getting into the cache, therefore, when sending the blocks of events serially, the order of events for a single measured location (client or service) remains the same. Between two different measurement locations, the correct ordering as well as the same time can not be guaranteed without taking into consideration the concrete implementation (e.g., the phase order in CXF). This would make the solution largely technology dependent and was not necessary, as most metrics do not directly measure between two separate measurement locations.

DSL Transformation

The transformation of the MDDSL is done at runtime, when an MD is sent to the monitor and it is used in an SLA. It is converted into a `MeasurementDescription` object, including its properties, a list of used MPs and the transformations necessary to compute the metric. It can only be converted if it is used in an SLA, because the transformations include the service for which the measurement should be taken.

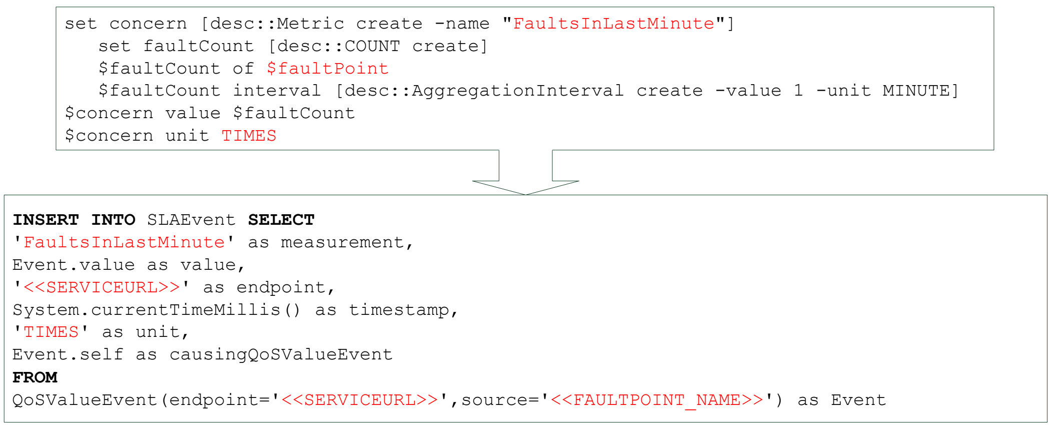
Transformation of the MPs is relatively straightforward. Every parameter of the MP is mapped to the corresponding attribute of an `MeasuringPoint` object. As the definition of the MPs is also defined in Frag, some validation can be done before the translation process. An important part of the translation is the generation of the name. Based on its parameters, its type and the address of the service where it is measured (which was obtained from the SLA), the name is generated. This name is sent with every measurement to be able to correlate the measurements to the MPs. Structure and examples of the names of the currently supported MPs can be seen in Figure 5.11.

One of the generated transformation transforms the events of the top-most function to `SLAEvents` (see Figure 5.12)



```
AMtP[<<SERVICEURL>>|<<LOCATION>>|<<CUSTOMNAME>>]
AMtP[http://127.0.0.1/svA|SERVER|demoAtMP]
AMvP[<<SERVICEURL>>|<<LOCATION>>|<<CUSTOMNAME>>]
AMvP[http://127.0.0.1/svA|SERVER|demoAvMP]
RMP[<<SERVICEURL>|<<LOCATION>>|<<INTERVAL>>]
RMP[http://127.0.0.1/svA|SERVER|5 MINUTES]
PMP[<<SERVICEURL>|<<CHAIN>>|<<PHASE>>|<<ISFAULT>>|AFT|<<AFTERLIST>>|BEF|<<BEFORELIST>>]
PMP[http://127.0.0.1/svA|SERVER_IN|receive|NORMAL|AFT|int.CeptA,int.CeptB|BEF|int.CeptC]
```

Figure 5.11: MP Names and Examples



```
set concern [desc::Metric create -name "FaultsInLastMinute"]
   set faultCount [desc::COUNT create]
   $faultCount of $faultPoint
   $faultCount interval [desc::AggregationInterval create -value 1 -unit MINUTE]
$concern value $faultCount
$concern unit TIMES
```

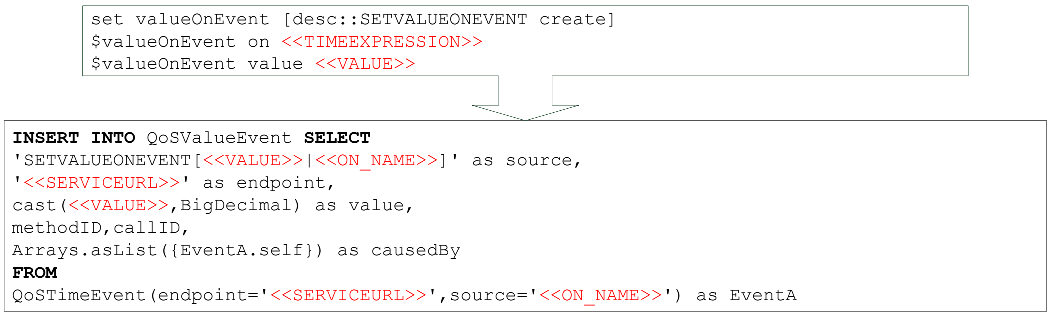
```
INSERT INTO SLAEvent SELECT
'FaultsInLastMinute' as measurement,
Event.value as value,
'<<SERVICEURL>>' as endpoint,
System.currentTimeMillis() as timestamp,
'TIMES' as unit,
Event.self as causingQoSValueEvent
FROM
QoSValueEvent(endpoint='<<SERVICEURL>>',source='<<FAULTPOINT_NAME>>') as Event
```

Figure 5.12: Transformation from the Uppermost Function to a Measurement (`SLAEvent`)

The transformation for functions follows a general scheme. For every function, there is at least one transformation. It creates events with a unique `source`. The `source` represents the name of the function which created the event. This name is build by coalescing the name of the function and its parameters. If one of these parameters is another function or MP, its name is used. The name has to be unique for this combination of parameters, to avoid duplicate event sources. Another important part of each transformation is the `causality` reference. Every transformation should add a reference to the causing event(s) to the event it creates, as described in Section 2.4.2. This makes is possible to detect causes of an SLA violation. Aggregation types, as mentioned above, are supported for many functions. They determine if the resulting event is calculated from events which share a common call (`SAME_CALL`), common method

(SAME_METHOD) or just their source and type. The aggregation type is propagated when a function uses data from another function, for example if a SUM function with SAME_METHOD aggregation uses a TIMEBETWEEN function as an event source (of parameter), the TIMEBETWEEN function also uses SAME_METHOD. When inheriting aggregation types, the SAME_CALL type is stronger than SAME_METHOD, which in turn is stronger than an unset aggregation type, meaning that a stronger type can overwrite a weaker inherited type. SAME_CALL aggregation is always a SAME_METHOD aggregation as well. Call-IDs and method-IDs are sent with the events and functions try to forward them, but that is not always possible when not using any aggregation type. There are two types of intervals: AggregationIntervals and TimeIntervals. Both have a value and a unit and are used to describe the size of a viewing window (e.g., for the size of the window, a SUM function is using). TimeIntervals only support time spans: milliseconds, seconds, minutes, hours and days. AggregationIntervals support a certain event count by using the events-unit.

As every function is transformed differently, the following list shows the use of every function and the corresponding transformation. Red text in the transformation declares data which is either filled in from the MDDSL instantiation (if it has the same name), or otherwise calculated from it. *SERVICEURL* is always determined from the SLA and *AGGREGATION* is inherited (see above) or overwritten with a stronger aggregation type. *INTERVALs* are converted to *INTERVAL_ESPER*, using a Esper-understandable syntax, and *INTERVAL_NAME*, which is a general independent notation. Text for SAME_METHOD aggregation is shown in dark green. Text for SAME_CALL aggregation is shown in light green. Blue *ORs* show possible alternatives in the same line. Blue *JOINLISTs* symbolize joining a list using the included text as a template and the second parameter as a placeholder between the elements. The list elements are named like the list, without an attached *"_LIST"*. Their sources have an attaches *"_NAME"*. When joining hash tables, the used convention is that the hash table is a list of key-value pairs, where the prefix *"_KEY"* represents the key and *"_VALUE"* the value.



```
set valueOnEvent [desc::SETVALUEONEVENT create]
$valueOnEvent on <<TIMEEXPRESSION>>
$valueOnEvent value <<VALUE>>
```

```
INSERT INTO QoSValueEvent SELECT
'SETVALUEONEVENT[<<VALUE>>|<<ON_NAME>>]' as source,
'<<SERVICEURL>>' as endpoint,
cast(<<VALUE>>,BigDecimal) as value,
methodID,callID,
Arrays.asList({EventA.self}) as causedBy
FROM
QoSTimeEvent(endpoint='<<SERVICEURL>>',source='<<ON_NAME>>') as EventA
```

Figure 5.13: MDDSL Transformation of the SETVALUEONEVENT-Function

The SETVALUEONEVENT-function is used to create a ValueEvent when a TimeEvent is received. It does not support any aggregations, as it only has one event parameter.

```
set timeBetweenEvents [desc::INTERVAL create]
$timeBetweenEvents between <<TIMEEXPRESSION>>
```

```
INSERT INTO QoSValueEvent SELECT
'INTERVAL[<<BETWEEN_NAME>>|<<AGGREGATION>>]' as source,
'<<SERVICEURL>>' as endpoint,
cast(Math.abs(EventA.timestamp-prev(1,EventA.timestamp)),BigDecimal) as value,
EventA.methodID as methodID,
Arrays.asList({EventA.self}) as causedBy
FROM
QoSTimeEvent(endpoint='<<SERVICEURL>>',source='<<BETWEEN_NAME>>').std:groupwin(methodID)
.win:length(2) as EventA
HAVING NOT
prev(1,EventA.timestamp) IS NULL
```

Figure 5.14: MDDSL Transformation of the `INTERVAL` Function

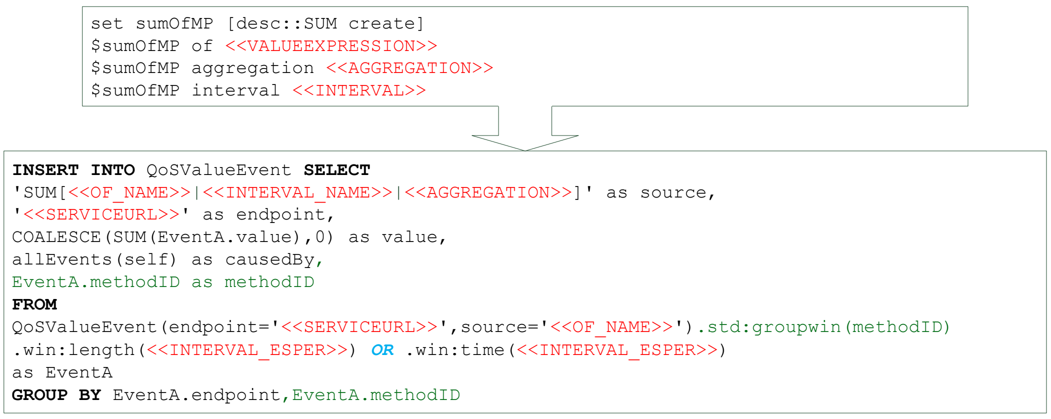
`INTERVAL` fires a `ValueEvent`, containing the time difference to the last event, every time a `TimeEvent` is received. The difference is calculated by subtracting the timestamps. It only supports `SAME_METHOD` aggregation.

```
set ifMP [desc::IF create]
$ifMP value <<VALUEEXPRESSION>>
$ifMP comparison "<<COMPARISON>>"
$ifMP then <<THENVALUE>>
$ifMP else <<ELSEVALUE>>
```

```
INSERT INTO QoSValueEvent SELECT
'IF[<<VALUE_NAME>>|<<COMPARISON_NAME>>|<<THENVALUE>>|<<ELSEVALUE>>]' as source,
'<<SERVICEURL>>' as endpoint,
(case when <<COMPARISON_ESPER>>
then cast(<<THENVALUE>>,BigDecimal)
else cast(<<ELSEVALUE>>,BigDecimal) end) as value,
callID,methodID, Arrays.asList({self}) as causedBy
FROM
QoSValueEvent(endpoint='<<SERVICEURL>>',source='<<VALUE_NAME>>')
```

Figure 5.15: MDDSL Transformation of the `IF` Function

The `IF`-function allows to create `ValueEvents` based on a `condition` involving another `ValueEvent`. It uses Esper's `CASE-WHEN` construct to implement the conditional. Its `comparison` parameter is currently directly used in Esper, using “value” as a variable for the event received through the `value` parameter. `IF` supports only one parameter, therefore it does not use any aggregation types.

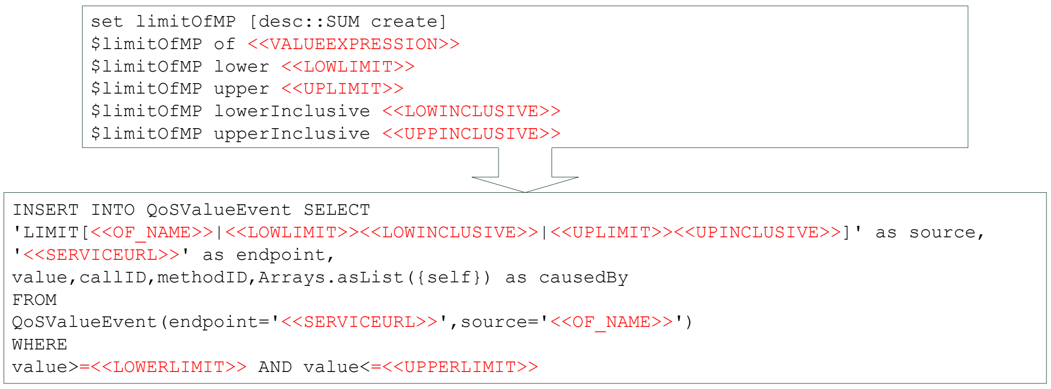


```
set sumOfMP [desc::SUM create]
$sumOfMP of <<VALUEEXPRESSION>>
$sumOfMP aggregation <<AGGREGATION>>
$sumOfMP interval <<INTERVAL>>
```

```
INSERT INTO QoSValueEvent SELECT
'SUM[<<OF_NAME>>|<<INTERVAL_NAME>>|<<AGGREGATION>>]' as source,
'<<SERVICEURL>>' as endpoint,
COALESCE(SUM(EventA.value),0) as value,
allEvents(self) as causedBy,
EventA.methodID as methodID
FROM
QoSValueEvent(endpoint='<<SERVICEURL>>',source='<<OF_NAME>>').std:groupwin(methodID)
.win:length(<<INTERVAL_ESPER>>) OR .win:time(<<INTERVAL_ESPER>>)
as EventA
GROUP BY EventA.endpoint,EventA.methodID
```

Figure 5.16: MDDSL Transformation of the `SUM`-Function

The `SUM`-function calculates the sum of all specified `ValueEvents` (`of`) in a certain sliding `interval`. The `SAME_CALL`-aggregation is not supported, because this would lead to the system waiting for future events for every past call, as the monitor does not know when a call is over. Depending on the type of interval, a `time`- or `length`-window is used. `SUM` uses the Esper `COALESQUE` function to return 0 if the specified interval gets empty. `allEvents()` is a custom Esper aggregation function to get all events from a window. The `self` property is a reference to the `QoSValueEvent` and was added for convenient access.

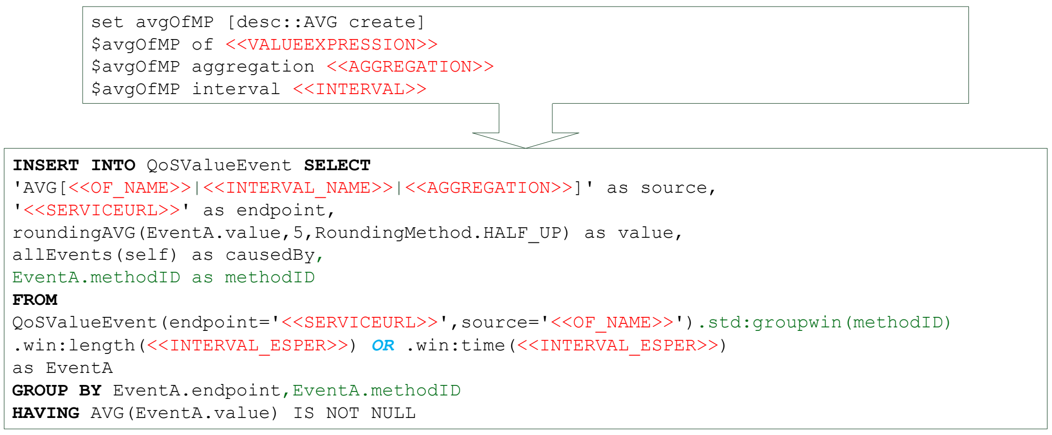


```
set limitOfMP [desc::SUM create]
$limitOfMP of <<VALUEEXPRESSION>>
$limitOfMP lower <<LOWLIMIT>>
$limitOfMP upper <<UPLIMIT>>
$limitOfMP lowerInclusive <<LOWINCLUSIVE>>
$limitOfMP upperInclusive <<UPPINCLUSIVE>>
```

```
INSERT INTO QoSValueEvent SELECT
'LIMIT[<<OF_NAME>>|<<LOWLIMIT>><<LOWINCLUSIVE>>|<<UPLIMIT>><<UPINCLUSIVE>>]' as source,
'<<SERVICEURL>>' as endpoint,
value,callID,methodID,Arrays.asList({self}) as causedBy
FROM
QoSValueEvent(endpoint='<<SERVICEURL>>',source='<<OF_NAME>>')
WHERE
value>=<<LOWERLIMIT>> AND value<=<<UPPERLIMIT>>
```

Figure 5.17: MDDSL Transformation of the `LIMIT` Function

`LIMIT` forwards `ValueEvents` only when their value is within in a certain interval. `LOW-INCLUSIVE` and `UPPINCLUSIVE` can be set to “exclusive” or “inclusive”, influencing the comparison in the transformation.

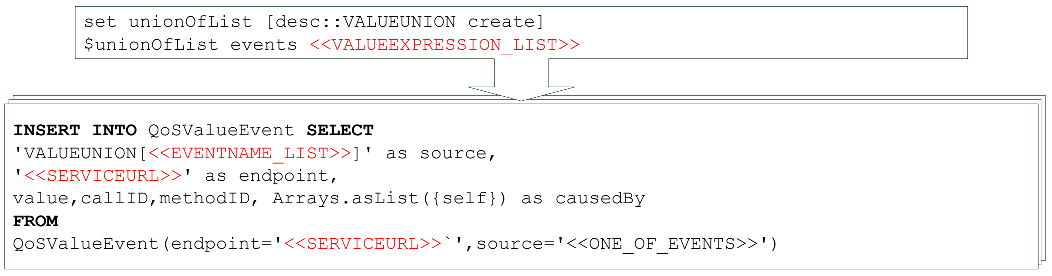


```
set avgOfMP [desc::AVG create]
$avgOfMP of <<VALUEEXPRESSION>>
$avgOfMP aggregation <<AGGREGATION>>
$avgOfMP interval <<INTERVAL>>
```

```
INSERT INTO QoSValueEvent SELECT
'AVG[<<OF_NAME>>|<<INTERVAL_NAME>>|<<AGGREGATION>>]' as source,
'<<SERVICEURL>>' as endpoint,
roundingAVG(EventA.value,5,RoundingMethod.HALF_UP) as value,
allEvents(self) as causedBy,
EventA.methodID as methodID
FROM
QoSValueEvent(endpoint='<<SERVICEURL>>',source='<<OF_NAME>>').std:groupwin(methodID)
.win:length(<<INTERVAL_ESPER>>) OR .win:time(<<INTERVAL_ESPER>>)
as EventA
GROUP BY EventA.endpoint,EventA.methodID
HAVING AVG(EventA.value) IS NOT NULL
```

Figure 5.18: MDDSL Transformation of the `AVG` Function

`AVG` works essentially the same as `SUM`. The only two difference is, that `AVG` does not output any 0-valued events if the interval gets empty. It uses a custom, `roundedAVG` aggregation function, because the original version throws `ArithmeticExceptions` if a `BigDecimal` value can not be expressed in decimal notation. The new function uses rounding in such cases.

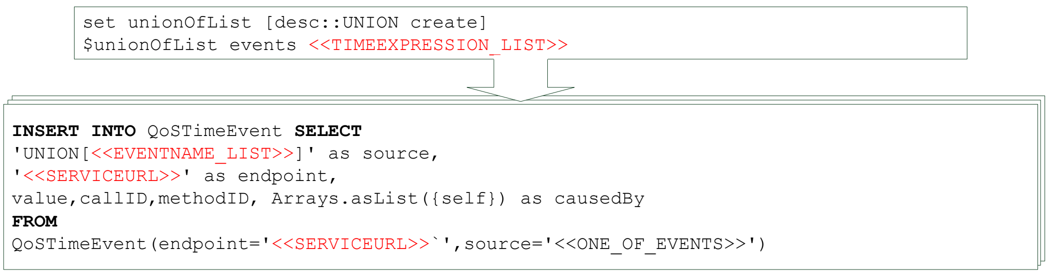


```
set unionOfList [desc::VALUEUNION create]
$unionOfList events <<VALUEEXPRESSION_LIST>>
```

```
INSERT INTO QoSValueEvent SELECT
'VALUEUNION[<<EVENTNAME_LIST>>]' as source,
'<<SERVICEURL>>' as endpoint,
value,callID,methodID, Arrays.asList({self}) as causedBy
FROM
QoSValueEvent(endpoint='<<SERVICEURL>>`',source='<<ONE_OF_EVENTS>>')
```

Figure 5.19: MDDSL Transformation of the `VALUEUNION` Function

`VALUEUNION` is an example for an MDDSL function, which creates multiple transformations. For every event in the `events` list, a transformation is created which forwards the events and sets them to a single source, effectively joining the event streams. *ONE_OF_EVENTS* contains one event of the list for each transformation. *EVENTNAMES_LIST* is a comma separated list of the names of the events' sources.

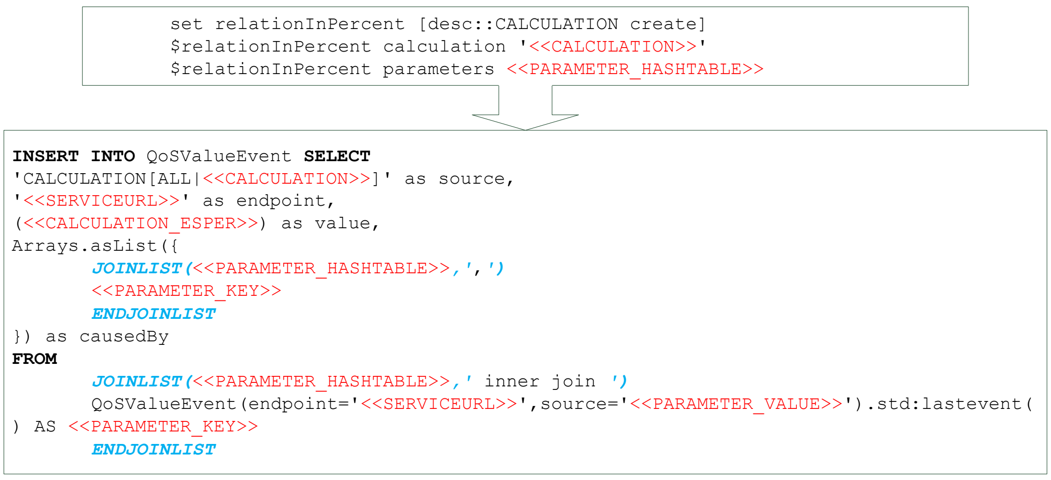


```
set unionOfList [desc::UNION create]
$unionOfList events <<TIMEEXPRESSION_LIST>>
```

```
INSERT INTO QoSTimeEvent SELECT
'UNION[<<EVENTNAME_LIST>>]' as source,
'<<SERVICEURL>>' as endpoint,
value,callID,methodID, Arrays.asList({self}) as causedBy
FROM
QoSTimeEvent(endpoint='<<SERVICEURL>>`',source='<<ONE_OF_EVENTS>>')
```

Figure 5.20: MDDSL Transformation of the `UNION` Function

`UNION` works exactly as `VALUEUNION`, except it uses and creates `TimeEvents` instead of `ValueEvents`.

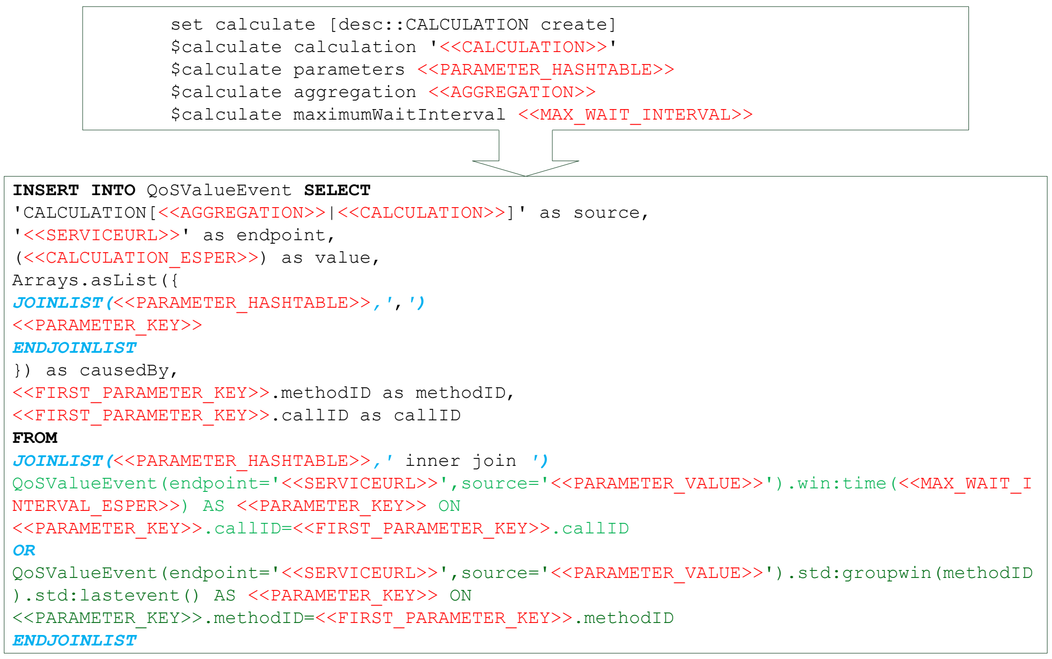


```
        set relationInPercent [desc::CALCULATION create]
        $relationInPercent calculation '<<CALCULATION>>'
        $relationInPercent parameters <<PARAMETER_HASHTABLE>>
```

```
INSERT INTO QoSValueEvent SELECT
'CALCULATION[ALL|<<CALCULATION>>]' as source,
'<<SERVICEURL>>' as endpoint,
(<<CALCULATION_ESPER>>) as value,
Arrays.asList({
       JOINLIST(<<PARAMETER_HASHTABLE>>,',')
       <<PARAMETER_KEY>>
       ENDJOINLIST
}) as causedBy
FROM
       JOINLIST(<<PARAMETER_HASHTABLE>>,' inner join ')
       QoSValueEvent(endpoint='<<SERVICEURL>>',source='<<PARAMETER_VALUE>>').std:lastevent(
) AS <<PARAMETER_KEY>>
       ENDJOINLIST
```

Figure 5.21: MDDSL Transformation of the `CALCULATION` Function

The transformation for the `CALCULATION`-function is different whether it is using an aggregation type or not. Both cases are visualized separately. They use the `calculation` attribute and the `parameters` hash map to do calculations. In `calculation`, the parameter keys are substituted with the sources of the corresponding events, always using the latest received values for each source. *FIRST_PARAMETER_KEY* refers to the key of the first parameter.

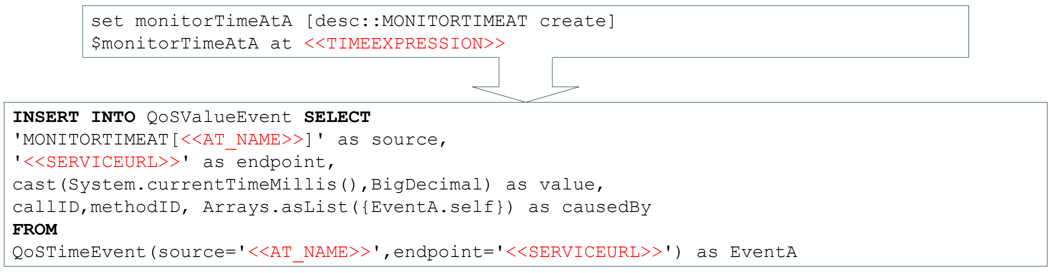


```
        set calculate [desc::CALCULATION create]
        $calculate calculation '<<CALCULATION>>'
        $calculate parameters <<PARAMETER_HASHTABLE>>
        $calculate aggregation <<AGGREGATION>>
        $calculate maximumWaitInterval <<MAX_WAIT_INTERVAL>>
```

```
INSERT INTO QoSValueEvent SELECT
'CALCULATION[<<AGGREGATION>>|<<CALCULATION>>]' as source,
'<<SERVICEURL>>' as endpoint,
(<<CALCULATION_ESPER>>) as value,
Arrays.asList({
JOINLIST(<<PARAMETER_HASHTABLE>>,',')
<<PARAMETER_KEY>>
ENDJOINLIST
}) as causedBy,
<<FIRST_PARAMETER_KEY>>.methodID as methodID,
<<FIRST_PARAMETER_KEY>>.callID as callID
FROM
JOINLIST(<<PARAMETER_HASHTABLE>>,' inner join ')
QoSValueEvent(endpoint='<<SERVICEURL>>',source='<<PARAMETER_VALUE>>').win:time(<<MAX_WAIT_I
NTERVAL_ESPER>>) AS <<PARAMETER_KEY>> ON
<<PARAMETER_KEY>>.callID=<<FIRST_PARAMETER_KEY>>.callID
OR
QoSValueEvent(endpoint='<<SERVICEURL>>',source='<<PARAMETER_VALUE>>').std:groupwin(methodID
).std:lastevent() AS <<PARAMETER_KEY>> ON
<<PARAMETER_KEY>>.methodID=<<FIRST_PARAMETER_KEY>>.methodID
ENDJOINLIST
```

Figure 5.22: MDDSL Transformation of the `CALCULATION` Function With Aggregation

If the aggregation type is set, another transformation is used for the `CALCULATION`-function. The blue *OR* shows that grouping by call or method in this case are *alternatives*. When grouping by call, a time interval is needed, to prevent infinite waiting. It is set through the `maximumWaitTime` property.

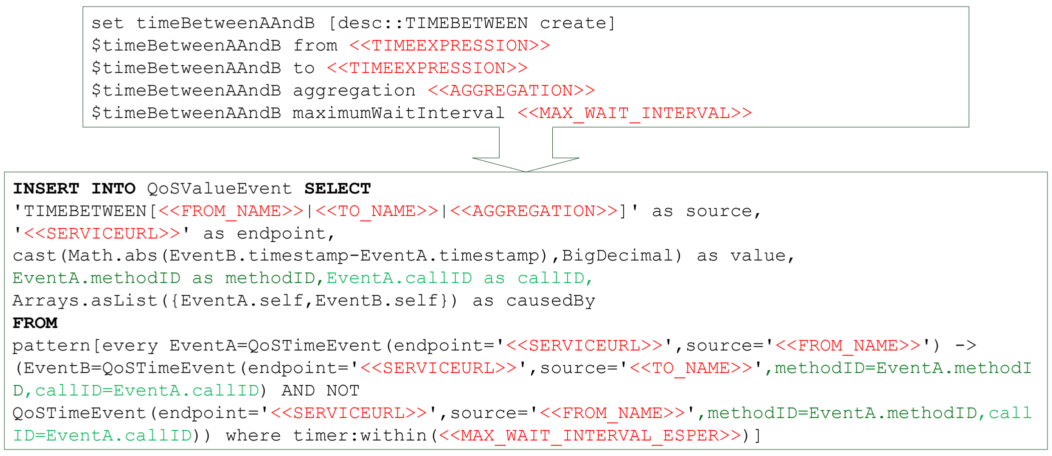


```
set monitorTimeAtA [desc::MONITORTIMEAT create]
$monitorTimeAtA at <<TIMEEXPRESSION>>
```

```
INSERT INTO QoSValueEvent SELECT
'MONITORTIMEAT[<<AT_NAME>>]' as source,
'<<SERVICEURL>>' as endpoint,
cast(System.currentTimeMillis(),BigDecimal) as value,
callID,methodID, Arrays.asList({EventA.self}) as causedBy
FROM
QoSTimeEvent(source='<<AT_NAME>>',endpoint='<<SERVICEURL>>') as EventA
```

Figure 5.23: MDDSL Transformation of the `MONITORTIMEAT` Function

`MONITORTIMEAT` is a simple function, used to calculate the time at which a certain `TimeEvent` is processed at the monitor.

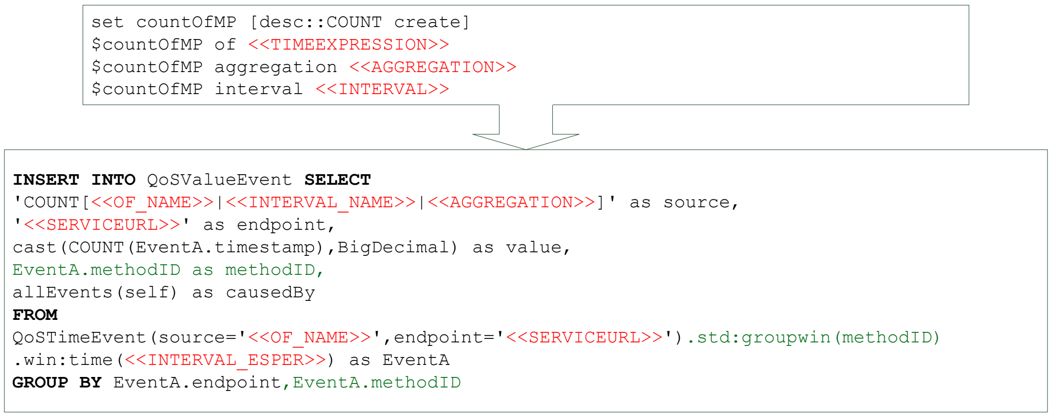


```
set timeBetweenAAndB [desc::TIMEBETWEEN create]
$timeBetweenAAndB from <<TIMEEXPRESSION>>
$timeBetweenAAndB to <<TIMEEXPRESSION>>
$timeBetweenAAndB aggregation <<AGGREGATION>>
$timeBetweenAAndB maximumWaitInterval <<MAX_WAIT_INTERVAL>>
```

```
INSERT INTO QoSValueEvent SELECT
'TIMEBETWEEN[<<FROM_NAME>>|<<TO_NAME>>|<<AGGREGATION>>]' as source,
'<<SERVICEURL>>' as endpoint,
cast(Math.abs(EventB.timestamp-EventA.timestamp),BigDecimal) as value,
EventA.methodID as methodID,EventA.callID as callID,
Arrays.asList({EventA.self,EventB.self}) as causedBy
FROM
pattern[every EventA=QoSTimeEvent(endpoint='<<SERVICEURL>>',source='<<FROM_NAME>>') ->
(EventB=QoSTimeEvent(endpoint='<<SERVICEURL>>',source='<<TO_NAME>>',methodID=EventA.methodI
D,callID=EventA.callID) AND NOT
QoSTimeEvent(endpoint='<<SERVICEURL>>',source='<<FROM_NAME>>',methodID=EventA.methodID,call
ID=EventA.callID)) where timer:within(<<MAX_WAIT_INTERVAL_ESPER>>)]
```

Figure 5.24: MDDSL Transformation of the `TIMEBETWEEN` Function

The `TIMEBETWEEN`-function is used in most measurements and measures the time between two TimeEvents. The function uses the last (for every aggregation) received `from` event to determine the time a following `to` event. In case of a new `from` event, the timer is reset. The `maximumWaitInterval` uses a `TimeInterval` to define a maximum time to wait for the second event. This prevents waiting for events which are lost on the way or maliciously omitted.



```
set countOfMP [desc::COUNT create]
$countOfMP of <<TIMEEXPRESSION>>
$countOfMP aggregation <<AGGREGATION>>
$countOfMP interval <<INTERVAL>>
```

```
INSERT INTO QoSValueEvent SELECT
'COUNT[<<OF_NAME>>|<<INTERVAL_NAME>>|<<AGGREGATION>>]' as source,
'<<SERVICEURL>>' as endpoint,
cast(COUNT(EventA.timestamp),BigDecimal) as value,
EventA.methodID as methodID,
allEvents(self) as causedBy
FROM
QoSTimeEvent(source='<<OF_NAME>>',endpoint='<<SERVICEURL>>').std:groupwin(methodID)
.win:time(<<INTERVAL_ESPER>>) as EventA
GROUP BY EventA.endpoint,EventA.methodID
```

Figure 5.25: MDDSL Transformation of the `COUNT` Function

`COUNT` is an aggregation function, similar to `SUM` and `AVG`, which calculates the number of events in a certain interval. As it counts events, the only supported `interval` is a `TimeInterval`. Other than that, it has the same restrictions.

```
set countOfMP [desc::TUMBLING_COUNT create]
$countOfMP of <<TIMEEXPRESSION>>
$countOfMP aggregation <<AGGREGATION>>
$countOfMP interval <<INTERVAL>>
```

```
INSERT INTO QoSValueEvent SELECT
'TUMBLING_COUNT[<<OF_NAME>>|<<INTERVAL_NAME>>]' as source,
'<<SERVICEURL>>' as endpoint,
cast(COUNT(*),BigDecimal) as value,
allEvents(self) as causedBy
FROM
QoSTimeEvent(source='<<OF_NAME>>',endpoint='<<SERVICEURL>>')
.win:time_batch(<<INTERVAL_ESPER>>,'FORCE_UPDATE') as EventA
```

Figure 5.26: MDDSL Transformation of the `TUMBLING_COUNT` Function

`TUMBLING_COUNT` does the same as `COUNT`, except it uses a tumbling window, clearing all events at the end of an interval. It also sends events if the interval is empty, but does not support any aggregation type.

```
set onChange [desc::ONEVENTCHANGE create]
$onChange events <<EVENT_LIST>>
$onChange fireOn <<FIREON_LIST>>
```

```
INSERT INTO QoSTimeEvent SELECT
'ONEVENTCHANGE[
        JOINLIST(<<FIREON_LIST>>,',')
                <<FIREON_NAME>>
        ENDJOINLIST | JOINLIST(<<EVENT_LIST>>,',')
                <<EVENT_NAME>>
        ENDJOINLIST
]' as source,
endpoint,timestamp,callID,methodID,Arrays.asList({self,prev(1,self)}) as causedBy
FROM
QoSTimeEvent(endpoint='<<SERVICEURL>>',
        JOINLIST(<<EVENT_LIST>>,'OR')
                source='<<EVENT_NAME>>'
        ENDJOINLIST
).win:length(2)
WHERE
NOT prev(1,source)=source AND (
        JOINLIST(<<FIREON_LIST>>,'OR')
                source='<<FIREON_NAME>>'
        ENDJOINLIST
)
```

Figure 5.27: MDDSL Transformation of the `ONEVENTCHANGE` Function

The `ONEVENTCHANGE`-function observes a stream, consisting of every event from the `events`-list. It fires if it detects an event from the `fireOn`-list, but only if the last received event was not from the same source.

6. Implementation

To be able to demonstrate and evaluate the concept, a prototype was implemented. This chapter describes the implementation of this prototype, following a similar structure to the design chapter. The first section describes how the components of the system were implemented, based on the general design. Afterwards, a section describes the concrete structure of the DSL transformation process. Finally, a section is dedicated to general problems which occurred while implementing the presented concept with the technologies used in the prototype.

The prototype is programmed in Java, using Apache CXF version 2.3.6 as its Web service framework. CXF is also used for MP distribution and as an administration interface for the monitor. The currently used MLF is able to measure AMPs(AtMPs and AvMPs), RMPs and PMPs in CXF. The events are sent to the monitor via Java RMI at the moment. As predetermined by the DSL design, the prototype uses Frag as a framework for DSL transformation. Esper is used as the CEP engine for internal event processing. The `Web interface` is implemented as a JavaServer Faces (JSF) [53] Web application. All permanent data is stored in a MySQL [77] database.

6.1 Components

This section details the implementation of the prototype. The monitor implementation is followed by an implementation of the MLF for the Apache CXF framework. Lastly, the Web interface is presented.

6.1.1 Monitor

The monitor is the core of the monitoring system. It manages the SLA, the MDs and the services which should be monitored. It also distributes the MPs to the services and collects the measurements to combine them to metrics. The metrics are used to check for SLA violations and, if necessary, send notifications.

Interface Implementation

In our prototype the `Measurement Interface` and the `Query Interface` are combined into a single `IQoSMonitor` interface. As SLA and MDs are sent to the interface as DSL strings, the interface uses the `DSLResolver` to convert it to Java objects. It then forwards all requests to the corresponding systems, while taking care of locks to avoid race conditions. In addition, it also saves the accepted DSL in the database for permanent storage. It obtains a lock to the event system and forwards the measurements. For performance reasons, the `Measurement Interface` is implemented using Java RMI.

DSLResolver

The `DSLResolver` is used to convert the MDDSL and SLADSL statements to Java objects. For every translation request it initializes a Frag interpreter and sends the DSL string. Frag interprets it, but the result are `FragObjects`, which can not be easily read in Java. Therefore the `DSLResolver` transforms the structure into a Java object model. Some semantic verification is done by the resolver, using additional information sent by the interface. That way, for example, it can be checked if the SLA only uses existing services and metrics which are defined by MDs. The detailed transformation process is described in Section 6.2

EventEngine

The `EventEngine` wraps, initializes and configures the Esper system. The rest of the system can add groups of `Transformation` objects, which wrap an EPL string. Queries are implemented through so-called `EventEnginePlugins`. Other subsystems in the monitor can register such plugins to be notified of certain events or to add transformations without the use of `Transformation` objects.

The engine differentiates between transformations and transformations groups (simply a list of transformations). Transformation groups are used to group transformations which belong to each other, such as the transformations generated from a single MD. Any entity in the system can add a transformation group and retrieve an ID which identifies it. This makes it easy to manage it and to delete it from the `EventEngine` if necessary. To avoid duplicate transformations (not transformation groups), a special hash map is used, which has a reference counter associated with each transformation. This way, duplicates can be avoided and the transformation is only deleted when the last reference is removed. Because of the automatic generation of the transformations and the special design of the DSL translation, it can be assumed that transformations which produce events with the same source parameter also have the same syntax. This makes it easier to detect duplicates. Instead of elimination duplicates, the event names could also be unique for every transformation instead of being unique for the way an event is generated, but by avoiding duplicate events, a performance benefit is gained, especially when using many similar measurements in one service.

The plugins are implemented by extending the `EventEnginePlugin` and implementing the `initialize()`-function. This function is called when the plugin is added and provides access to the `EventEngine`. Plugins can then add transformations or register themselves as observers to an EPL query, submitted through the `registerQuery()` function.
Currently implemented plugins are:

- `HistoryPlugin`: A plugin, which saves the measured metrics to the MySQL database. The query only outputs events every five seconds and all events received in this time span are added to the database as a batch, for performance optimization.
- `ConsoleOutputPlugin`: A debug plugin which outputs all events to the console, as they occur.

```
@ActionPlugin(scheme="mailto",simpleName="EMail−Action")
public class EMailActionPlugin extends ActionSystemPlugin
```

Listing 6.1: Example Action Plugin Annotation

- `QoSActionPlugin`: This plugin checks for `ActionEvents`, which are generated by the constraint systems transformations. If an `ActionEvent` is detected, it notifies the `Action System`.

Measurement System

The `Measurement System` gets MDs as Java objects and stores them. These MD objects contain the metric name, the service URI, a list of used MPs and a set of `Transformation` objects. To add an MD, it is split into the MP list and the transformations. While the MPs are sent to the corresponding services via Web service calls, the MDs are stored at the monitor, separately for each service. This is done through synchronization, only adding new MDs and deleting old ones. If an MD has been changed, the MDs are synchronized by name, including all services. If an SLA has changes, they are synchronized by service. The transformations are added the the `EventEngine`. The resulting IDs are stored with the MDs of a service, to be able to remove the transformations from the `EventEngine`, when the MD is changed or no longer used.

Constraint System

The `Constraint System` receives `Constraint` Java objects from the `Management Interface`. It transforms them to two types of EPL transformations: `ConstraintEvent` transformations and `ActionEvent` transformations. Details on this transformation process can be found in Subsection 6.2.1. They are added to the `EventEngine`, storing the transformation IDs and the constraints themselves. If new constraints are set, they are synchronized to the existing constraints, adding new ones and removing old ones.

Action System

The `Action System` has two parts: the `ActionPlugin` in the `EventEngine` and the `ActionSystem`. As mentioned before, the `ActionPlugin` waits for `ActionEvents` and sends them to the `ActionSystem`. The `ActionEvent` contains information about the `ActionPlugin` which should be executed as well as a parameter.

Plugins are registered through the `registerPlugin()` function at system startup. They have to extend the class `ActionSystemPlugin` and have an `ActionPlugin` annotation, which determines how the plugin can be referenced in an SLA (see Listing 6.1).

```
DB_URI=jdbc:mysql://localhost/QoS
DB_USERNAME=__USERNAME__
DB_PASSWORD=__PASSWORD__
MONITOR_MANAGEMENT_ADDRESS=http://192.168.100.1:10000/qos-monitor
MONITOR_MEASUREMENT_ADDRESS=//192.168.100.1/QoSMonitor
```

Listing 6.2: Configuration File (monitor.properties) for the Monitor

There are two possibilities for implementing an `ActionSystemPlugin`:

- By implementing ``` ``notify(String address,ActionEvent event)'' ```.
- By implementing ``` ``notify(String address,String content)'' ``` and calling `super(String templatePath)` in the constructor, with the path to a Apache Velocity [8] template, which is used to transform the ActionEvent to text. In this template, `event` references the ActionEvent and `util` references an utility whose `urlEscape` method can be used to transform text to an URL compatible encoding. If no path is specified in the call to super(), a default template is used which contains all information from the ActionEvent (see Subsection B.4.1).

When interpreting an ActionEvent, an important part is causality. The causing `ConstraintEvent` can be retrieved by getting the `causingConstraintEvent` attribute.
The `SLAEvents` (containing the metric values) which caused the `ConstraintEvent` can be retrieved through the `causingSLAEvents` attribute. Each `SLAEvent` has a causing `QoSValueEvent` (`causingQoSValueEvent`). From then on, the causing event tree can be traversed using the `causedBy` attribute, which contains a list of `QoSValueEvents` or `QoSTimeEvents`.

Configuration

The configuration works via a "monitor.properties"-file. It contains access data for the MySQL database, a `MONITOR_MEASUREMENT_ADDRESS` and a `MONITOR_MANAGEMENT_ADDRESS` parameter for the location of the interfaces of the monitor:

6.1.2 Measuring Location Framework

For every framework that is supported, there must be an MLF implementation and an MPDSL document. The current prototype supports the Apache CXF framework and the three MPs, presented in Subsection 5.2.1. It basically consists of tree parts: the `MP Management and Measuring Subsystem`, the *Sending Subsystem* and the *System Integration*. Adding a message to the *Sending Subsystem* does not send it but waits for a send command. This allows grouped and delayed sending of measurements.

MP Management Subsystem

The `MP Management Subsystem` is the core of the MLF. The main work of the system is done by the `QoSManagers`. For every provided or used service, either a `ServerQoS-Manager` or a `ClientQoSManager` is created. The `ServerQoSManager` also creates the interface for receiving MP updates, if not already created by another ServerQoSManager. The `ClientQoSManager` is also used to query the monitor for MP updates.

When a new MP is received by any `QoSManager`, an object is created which can measure this kind of MP. Both, the MP and the object are stored in a table for easy lookup. The current prototype supports three MPs, which were defined in Subsection 5.2.1:

PMPs - PhaseMeasuringPoints : To attach a PMP, a `QoSInterceptor` is created. It is added to the chain and phase which is specified in the PMP and before respectively after the specified interceptors. If the PMP which corresponds to the interceptor has its `ISFAULT` property set, it fires if its `handleFault`-method is called. Otherwise it fires if its `handleMessage`-method is called.

Every measurement contains a `method-ID` and a `call-ID`. The method-ID can be queried from the CXF framework at the client and the service, but the `call-ID` has to be generated for every call. A problem occurring it, that the `call-IDs` at the service and the client have to be synchronized. Our framework design solves this by sending the `call-ID` with the message, but the content of the message is not known at any phase in the message processing process. To solve this, multiple `IDInterceptors` are used. When a client sends a message, an `ID-Interceptor` creates a permanent `call-ID`. When the message is received at any other location, a temporary ID is created and assigned to the call. Later, when the message has been unwrapped, the `call-ID` is read from the header and assigned to the message. To avoid wrong IDs, all events with the old IDs have to be renamed in the sending system. This also means that messages with temporary IDs can not be sent until the have received their corrected, permanent `call-IDs`. The `Sending Subsystem` provides support for this.

RMPs - RepeatedMeasuringPoints : Every RMP is attached via a `RepeatedExecute-Measurement` object. It converts the interval value of the RMP from the specified value to milliseconds and uses a ScheduledThreadPoolExecutor to execute a task periodically. The task adds an event to the `Sending Subsystem` and immediately requests sending.

AMPs - ApplicationSpecificMeasuringPoints : To attach an AMP (AtMPs or AvMPs), the QoSManager just registers the MP in the table. This type of MP does not fire automatically but requires user intervention. The user can request a `ApplicationSpecificTimeMea-surement` or `ApplicationSpecificValueMeasurement` object (see Section 6.1.2). This object contains the functionality to add a new time- or value-event to the `Sending Sub-system` and request sending. Sending is only allowed if the AMP was attached previously. It is done through the `sendValue()` and `sendTime()` functions, depending on the AMP type.

```
ApplicationSpecificValueMeasurement ampValueSender;
ApplicationSpecificTimeMeasurement ampTimeSender;

ampValueSender=MonitoringSystem.getApplicationSpecificValueSender(
    "http://localhost:3000/demo", MeasuringPoint.SERVER, "FPS");
ampSender.sendValue(new BigDecimal(2)));
ampTimeSender=MonitoringSystem.getApplicationSpecificTimeSender(
    "http://localhost:3000/demo", MeasuringPoint.SERVER, "myTime");
ampSender.sendTime();
```

Listing 6.3: Using the Framework to Send AMP Data.

```
@Features (features = ‘‘qos.measuring.QoSMeasurementFeature’’)
```

Listing 6.4: Annotation for Registering the Monitored Service

Sending Subsystem

The `Sending Subsystem` is used for caching and sending events from MPs. It keeps all events in the right order through a queue. Sending is delayed to reduce sending overhead by sending the events grouped and because of the temporary `call-IDs` (see Section 6.1.2).

When an event is added to the `Sending Subsystem` it is cached in a queue. It supports changing the `call-IDs` of a group of events. The thread processes the queue until it reaches its end or an event which has not yet received its permanent `call-ID`. It then sends all events in one package to the monitor, preserving their order. If there are any more events in the queue, they are send later, after a predetermined delay.

In the current prototype, the events are send to the monitor using Java RMI. Future versions should rely on more platform independent methods (e.g., socket communication).

System Integration

The system is incorporated into an existing CXF application through the use of a `Feature` (see Section 4.1.2). To use it, a simple annotation has to be added to the Java interface or implementation class (see Listing 6.4). When the service is initialized, the Feature is also initialized, creating the necessary QoSManagers. If the Feature is added to the interface which the client uses for the proxy generation, it also uses the Feature.

To use application-specific measurements (through AMPs), the `Monitoring System` provides operations to receive objects which can be used for directly sending these measurements. The sender object can and should be reused, if needed. In the example (Listing 6.3), two senders are requested to send measurements to a service. The first sends `BigDecimal`-valued “FPS” information using the `sendValue()` function, the second sends a current timestamp using its `sendTime()` function.

```
QoSConfiguration.set("MONITOR_MANGEMENT_ADDRESS",
    "http://192.168.100.1:10000/qos-monitor");
QoSConfiguration.set("MONITOR_MEASUREMENT_ADDRESS",
    "//192.168.100.1/QoSMonitor");
QoSConfiguration.set("SENDING_IDLE_DELAY", "1000");
QoSConfiguration.set("MAX_BATCH_SEND", "150");

QoSConfiguration.set("DISTRIBUTION_INTERFACE_ADDRESS",
    "http://192.168.100.2:9000/qosservicemanager"); // only service
QoSConfiguration.set("MY_ADDRESS", "192.168.100.3"); //only client
```

Listing 6.5: Programmatic Configuration of the MLF

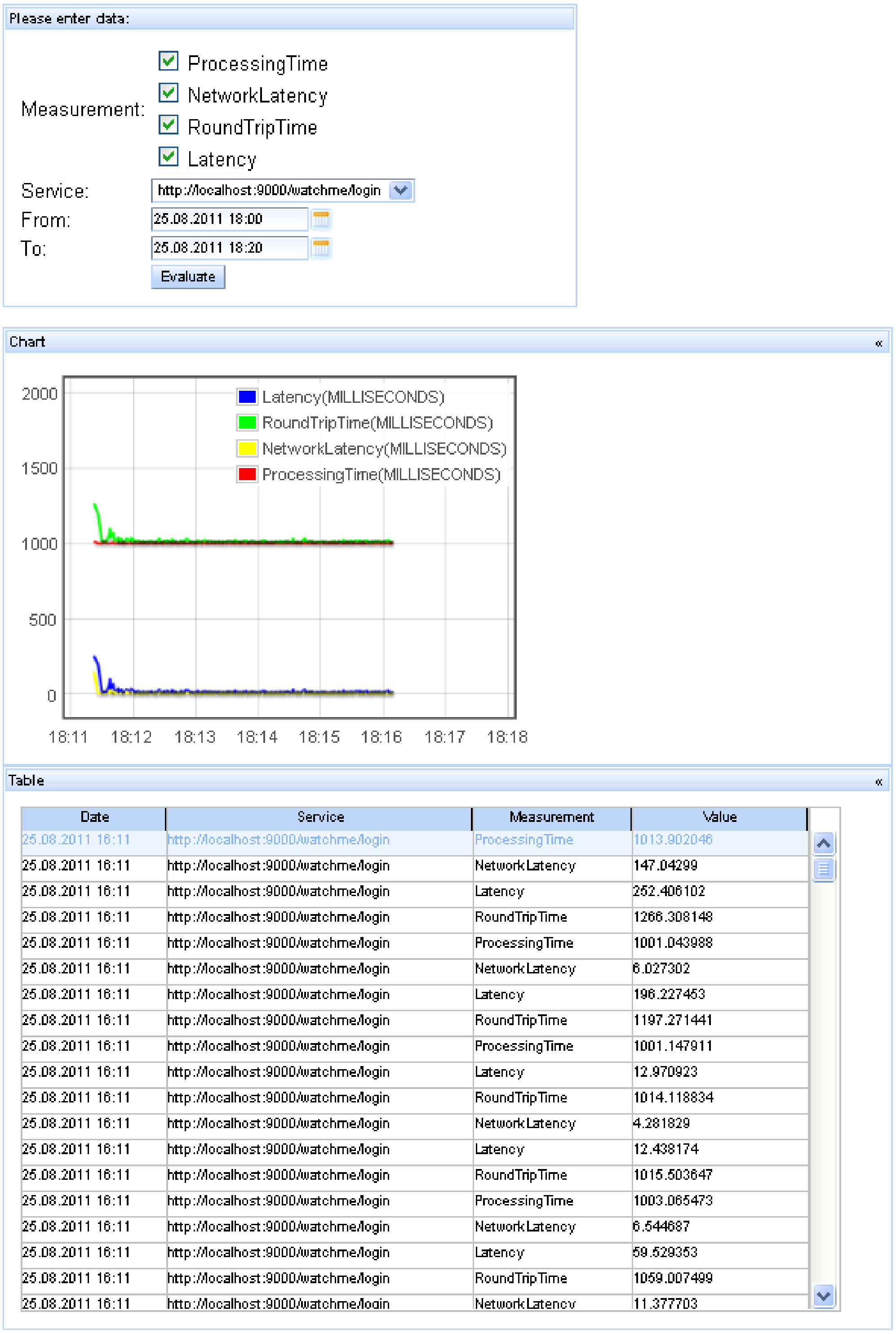
Configuration

The configuration for the MLF can be done programmatically through the `QoSConfiguration` class. Both, client and service, support the `MONITOR_MEASUREMENT_ADDRESS` property. It is used to specify the address of the monitor, for sending the measurements. They also support the `MONITOR_MANAGEMENT_ADDRESS` property, which is used to define the address of the monitor where current MPs can be queried. To configure the `Sending Subsystem`, the `SENDING_IDLE_DELAY` property is used to define the delay which should be waited if there are no new events to send. The `MAX_BATCH_SEND` property allows to specify a maximum number of events to send to the monitor in one batch. Limiting this value prevents a build-up of events without sending. A higher value improves performance by reducing the calls to the monitor, but delays the sending of the events. Services support a `DISTRIBUTION_INTERFACE_ADDRESS`, to specify the `Distribution Interface` address for this instance of the MLF. It is shared among all provided services. The client supports the `MY_ADDRESS` property, which can be used to specify an address for the client.

6.1.3 Web Interface

The `Web interface` is used to add services, MDs and SLAs to the monitor. It is simply a Web interface for the functionality provided by the monitors management and query Web services. All shown data is received from the monitor, and all changes are directly committed to the monitor. The interface is implemented as a Web application archive, using JSF. In addition it uses JBoss RichFaces 3.3.3 [89] and Facelets [52] for its UI components.

The Web application is separated into two parts. The first part is the search for *historical data* (past measurements). It allows to query and compare multiple metrics of multiple services over a certain time span. The result is shown in a zoomable and pannable graphical form and in a table (see Figure 6.1. The second part is the administration panel. It provides an overview of the current SLA, monitored services and registered MDs which can be used in the SLA. It also provides means to add, edit or delete them. Each editable component has its own add/edit page. Validity checks are done by the monitor and validation errors are shown on the pages.



Measurement:	✓ ProcessingTime ✓ NetworkLatency ✓ RoundTripTime ✓ Latency
Service:	http://localhost:9000/watchme/login
From:	25.08.2011 18:00
To:	25.08.2011 18:20

Date	Service	Measurement	Value
25.08.2011 18:11	http://localhost:9000/watchme/login	ProcessingTime	1013.902046
25.08.2011 18:11	http://localhost:9000/watchme/login	NetworkLatency	147.04299
25.08.2011 18:11	http://localhost:9000/watchme/login	Latency	252.406102
25.08.2011 18:11	http://localhost:9000/watchme/login	RoundTripTime	1266.308148
25.08.2011 18:11	http://localhost:9000/watchme/login	ProcessingTime	1001.043988
25.08.2011 18:11	http://localhost:9000/watchme/login	NetworkLatency	6.027302
25.08.2011 18:11	http://localhost:9000/watchme/login	Latency	196.227453
25.08.2011 18:11	http://localhost:9000/watchme/login	RoundTripTime	1197.271441
25.08.2011 18:11	http://localhost:9000/watchme/login	ProcessingTime	1001.147911
25.08.2011 18:11	http://localhost:9000/watchme/login	Latency	12.970923
25.08.2011 18:11	http://localhost:9000/watchme/login	RoundTripTime	1014.118834
25.08.2011 18:11	http://localhost:9000/watchme/login	NetworkLatency	4.281829
25.08.2011 18:11	http://localhost:9000/watchme/login	Latency	12.438174
25.08.2011 18:11	http://localhost:9000/watchme/login	RoundTripTime	1015.503647
25.08.2011 18:11	http://localhost:9000/watchme/login	ProcessingTime	1003.065473
25.08.2011 18:11	http://localhost:9000/watchme/login	NetworkLatency	6.544687
25.08.2011 18:11	http://localhost:9000/watchme/login	Latency	59.529353
25.08.2011 18:11	http://localhost:9000/watchme/login	RoundTripTime	1059.007499
25.08.2011 18:11	http://localhost:9000/watchme/login	NetworkLatency	11.377703

Figure 6.1: Screenshot of a Search Result in the Web Interface.

```
MONITOR_MANAGEMENT_ADDRESS=http://192.168.100.1:10000/qos-monitor
```
Listing 6.6: Configuration of the Web Interface (visualizer.properties)

Configuration

The interface must be configured in the file “WEB-INF/config/visualizer.properties”.
It supports a `MONITOR_MANAGEMENT_ADDRESS` parameter to specify the location of the monitor.

6.2 DSL Transformation

The transformation process varies, depending on the DSL. Generally, all used DSLs are interpreted by a instance of Frag.

The MPDSL uses the syntax of Frag and is interpreted by Frag at compile time. Its definitions are used later when an MD, using the MPDSL definitions, is transformed.

All other DSLs are used through the Java `DSLResolver` class. The resolver initializes a Frag interpreter and executes a common initialization code. This code includes files for the MPDSLs models and for the `esper-generator`, which is the main file for translating the MDDSL and SLADSL documents. The resolver then calls the needed methods of the `esper-generator`.

The DSLResolver has four methods:

- `SLA loadSLA(String slaCode,`
  `Map<String,String> serviceNameToUriMap`
  Translates the SLA code and returns a `SLA` Java object. The `serviceNameToUriMap` is used to translate the service names to its Uniform Resource Identifier (URI). It is derived from the service specification. To translate the SLA it calls the `loadSLA`-method of the `esper-generator`.

- `MeasurementInfo loadMeasurement(String measurementCode)`
  Loads the name and the unit of an MD in a `MeasurementInfo` object. To get the information it calls the `loadQoSConcern`-method of the `esper-generator`.

- `List<MeasurementDescription> resolveAllMDsForService(`
  `String serviceName, SLA sla,List<String> measurementCodes,`
  `Map<String, String> serviceNameToUriMap)`
  Loads all MDs for a specific service. The `sla` parameter is the Java object which was received from converting the SLA. It uses the `loadQoSConcern`-method of the `esper-generator` to load all measurements into memory and then uses the `generateMeasurementDescriptionsFromService`-method to get all translated MDs for this service.

- `List<MeasurementDescription> resolveMDsForAllUsingServices( String measurementCode, SLA sla, Map<String, String> serviceNameToUriMap)`
  Loads an MD for all services which are using it. It also uses the `loadQoSConcern`-method to load the measurement and then uses the `generateAllMeasurementDescriptions`-method to create a `MeasurementDescription` Java object for every service which uses this measurement.

6.2.1 SLADSL

If an SLA changes, the interface sends the new SLA to the `DSLResolver`. It initializes a Frag interpreter, loads the used measurements and lets Frag interpret the SLA using the "loadSLA" method of the esper-generator.

The SLADSL is an external DSL which is interpreted using Frag's DSL framework. At first, the `SLADSLParser` uses the prepared rules to parse the DSL text into tokens. The structure is rather simple, following the structure of the model. The conditions are parsed recursively. The tokens are then mapped, using the `SLADSLMapping`. In the process of mapping, all tokens are evaluated and inserted into the model at appropriate positions. For each type of token there is a special mapping. The mapping works recursively, where the currently active object is sent with the recursion and populated on the way. In addition to this filling of the model, there is a mixin for the condition which simplifies the detection of used measurements for a single condition.

When the `DSLResolver` receives the `SLA` Frag object, it parses them into simple Java objects, as the Frag objects are rather inconvenient to use. The result is returned to the interface. If necessary, the `Measurement and Distribution System` then updates the used MDs and notifies the affected services.

6.2.2 MDDSL

As the MDDSL is an internal DSL, it is written using the Frag language. The `Web interface` receives the MDs and forwards it to the monitor. The translation of the MDDSL mixes a lot with the SLADSL as the MDs are only fully interpreted if they are used in an SLA. If the SLA changes, all MDs in all services are updated. If an MD changes, only this metric is updated in every service it is used in. The `Measurement and Distribution System` of the monitor provides the necessary special synchronization methods.

To translate the MDDSL, the received text is sent to the `DSLResolver` which executes the `loadQoSConcern`-method of the `esper-generator`. The interpretation consists of executing the Frag code. This would pose a security risk in an open system and requires an additional authentication system. This was, however, out of the scope of this thesis and not crucial for demonstration of the general concept. The metric object is always named `"concern"`, so the engine can easily inspect it after execution. It is added to a global list of concerns for later use. The `loadQoSConcern`-method returns the name and the unit of the measurement.

A more concrete translation is done when an SLA is added, because the transformation needs to know which services the MD is assigned to. Both, `resolveAllMDsForService` and `generateAllMeasurementDescriptions` of the `esper-generator` use the `generateMDForService`-method. It uses a service's URI and an MD to generate a Frag object which contains the measurement name, the used service, the used MPs, and the used Esper transformations. The transformations and used MPs are generated through mixins of the MDDSL *Metric* object and functions and the MPs.

Beginning from the `Metric` object, the functions are called recursively until the MPs are reached. Every function and MP has a `generateEsper`-method, which is used to transform the function to EPL. Before creating the transformation, all used functions are transformed recursively. The generateEsper method has three parameters: `informationObj` is used to collect all needed transformations and used MPs, `aggregationObj` is used to inform the current function of its "inherited" aggregation status (see Section 5.3.3) and `service` holds the current service URI which is to be used. It returns the name of the function which is set as the source of events which are generated by its transformation. That way, a new function can use the name of its used functions to select its incoming events. The mixins of the different MPs only add a Frag object to the list of used MPs, which contains the MPs parameters and the generated name of the MP (see Section 5.3.3).

The resulting Frag objects are returned by the Frag interpreter, which converts them to Java objects, which are easier to handle. They are then then sent to the `Measurement and Distribution System`, synchronizing them with the currently used MPs.

7. Evaluation

This chapter contains an evaluation of the proposed solution from this thesis. At first, the performance and performance impact of the system are shown in a quantitative evaluation. Afterwards, a qualitative evaluations highlights various aspects of the system, like the expressiveness of the languages and trustability, in a critical discussion.

7.1 Quantitative Evaluation

The quantitative evaluation consists of various performance tests of the framework. Performance can be measured at two locations: at the MLF and at the monitor. The MLF has a direct performance influence on calls and the measurements of the framework. Therefore its impact has to be minimized. The calculations are done by the monitor, therefore the MLF's overhead lies in obtaining the data for the used MPs, and the sending to the monitor. As the monitor already queues the received events, the performance of the MLF can be increased by reducing the distance to the monitor.

The monitor itself only has an indirect influence on the measurements. If the monitor is overloaded, its response time can be low. This can delay the sending of events at the MLF which can ultimately lead to a build-up of events at the MLF, decreasing performance. Another influence is the used network capacity by the sent measurements.

7.1.1 Setup

The performance evaluation setup is composed of three computers in a 100mbit Ethernet network. Every computer uses the Sun JVM version 1.6.0_27-b07.

- The monitor runs on a Windows Vista PC, using an Intel Core 2 Quad Q660 at 2x2,4 Ghz with 3GB RAM.
- The measured service runs on a Windows 7 Laptop, using an Intel i3 M330 at 2x2,13 Ghz with 4GB RAM. The Jetty server of the Apache CXF framework is configured to use between 10 and 50 parallel threads.
- The clients run on a Window 7 PC, using an Intel Core 2 Duo E6600 at 2x2,4 Ghz with 2GB RAM. Depending on the test, a custom client, using the MLF or Apache JMeter [56], is used to generate the Web service calls.

The tests use the metrics which were presented in the motivating example (see Table 5.1), except for the application-specific metrics, because they are not calculated based on the Web service

Metric	PMPs	RMPs	AMPs	transformations
MTTR	0/2	0	0	6
Network Latency	3/2	0	0	6
MTBF	0/2	0	0	5
MTTF	0/2	0	0	6
Server Availability	0	1	0	5
Search Result Accuracy	0	0	1/1	5
Client Availability	0/2	0	0	7
Error Rate	0/2	0	0	7
Latency	3/2	0	0	6
Load	1/0	0	0	3
Proposition Quality	0	0	0/1	3
Round Trip Time	0/2	0	0	3
Processing Time	3/0	0	0	5
Total	10/16	1/0	1/2	67
Unique	6/5	1/0	1/2	60

Table 7.1: Comparison of the Metrics, the Generated MPs (Service/Client) and Transformations.

calls. Table Table 7.1 shows a comparison of the metrics, concerning the MPs they use and the transformations which are generated based on their description. The full text of the descriptions can be found in Section B.1.

The SLA which was chosen for these tests uses every metric, but makes sure that no actions fire, to keep the event log clean. It can be read in Section B.2.

Testing of every metric was achieved by implementing a test run which shuts down the service in certain intervals while the clients and the monitor keep running. This leads to a reproducible and testable scenario. To show all metrics, the complete system runs for 30 minutes, with three intentional service failures (three minutes after minute 5 and one minute after minutes 15 and 22). The client initiates 5 calls per second. The results are split into three graphs, according to their units and their purpose (see Figure 7.1). The number of events which are generated by the transformations, which ultimately build these measurements, is discussed in Subsection 7.1.3.

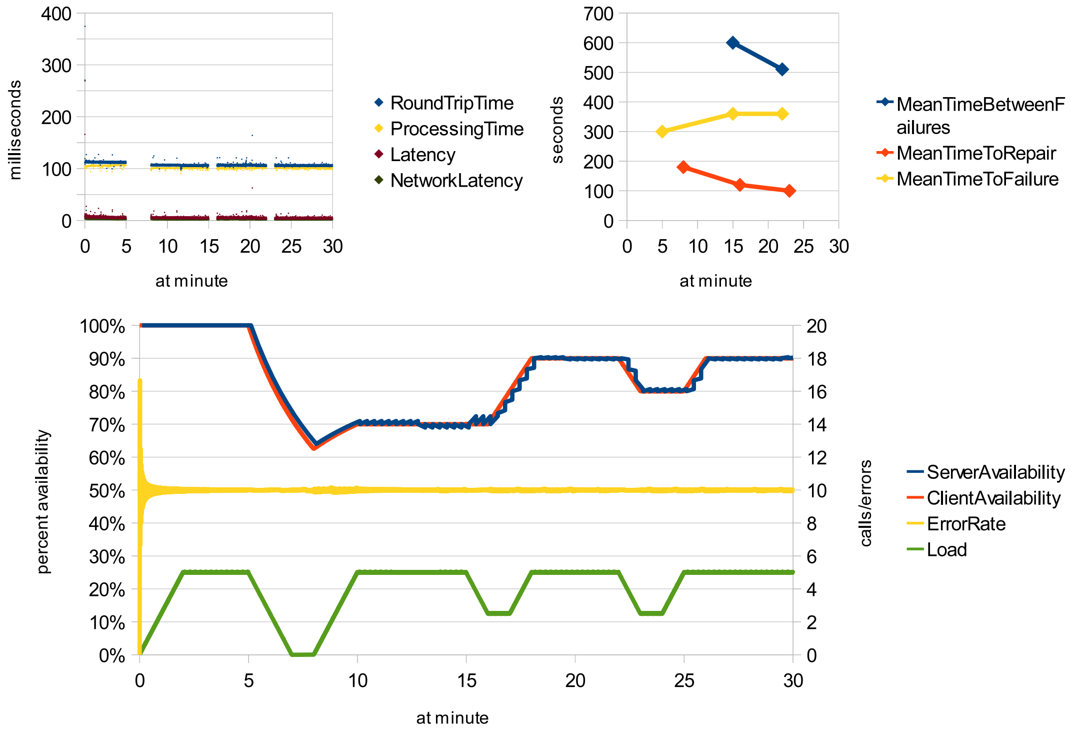


Figure 7.1: Example Measurements in Three Graphs

7.1.2 Performance of the MLF

One big problem when measuring service calls is the overhead the measuring introduces. Therefore this approach tries to minimize the measurement impact by (re)using interceptors and by using a separate monitor. Nonetheless, the interceptors and the process of sending the measurements to the monitor introduces an overhead. Therefore, this section evaluates the performance of the MLF at the client and the service. To avoid any influence by the event calculation, the monitor was set to receive events but not calculate any metrics. At the used configuration (using all shown metrics), the service has 8 unique PMPs (correspond to interceptors) and the client has 7 unique PMPs (see Table 7.1).

A first noticeable point is, that the measurement result using interceptors is not the same as with inline measurement, as the interceptors are located at a different position. Therefore a kind of accuracy can be computed when using the framework. The values were obtained by recording 10 calls per second for 10 minutes, while the processing time is about 100ms. Only the last minute was used to calculate the values, to avoid too much influence of the Java optimization, and the values are rounded to six decimal places. When comparing the inline round trip time to the framework-measured Round Trip Time, the inline measured time is always bigger than the framework-measured one. The mean of the differences lies at 0,476848 milliseconds with a variance of 0,006429 at the service computer. The inline processing time is always smaller than the one measured from the framework. The mean of the differences lies at 0,210954 milliseconds with a variance of 0,002547 at the client computer.

Different parts of the MLF have different influence on the measured results. The measurement itself, done by the interceptors, and the infrastructure for ID-generation and -alignment have a direct influence on the call duraction. The sending process runs in a separate thread and has an indirect influence.

To measure the overhead of the framework, one possibility is to calculate the overall time a call takes. The other possibility is to measure the overhead which is directly generated through the runtime of the interceptors.

Overall measurement

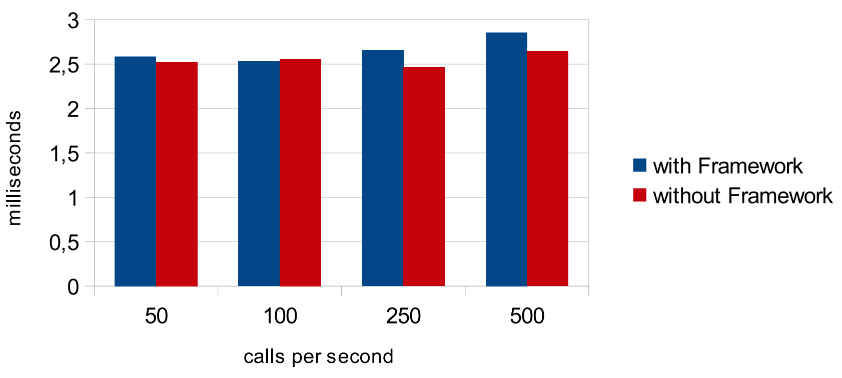


Figure 7.2: Performance of the MLF at the Service

To find the overhead of using the framework, the service is tested under different loads. The client essentially uses the same system but at much lower calls per second. At very low call rates (below 100 calls per second) the difference is not significant. Also measuring of the client is always influenced by the used service, therefore only the service is measured at high call rates.

Apache JMeter was used to generate the calls for the service performance evaluation. It rounds the call duration to milliseconds. In earlier tests Java optimizations led to a better performance when using high call frequencies. This could be reduced by using a warm-up period of three minutes before measuring for three minutes. In the warm-up period, as many calls as possible are sent to reduce the optimization influence. This procedure is done for the service with an activated MLF and without the MLF. Figure 7.2 shows that at low call rates the difference is not measurable with the used measurement technique. It can be seen that the overhead of using the framework is very small, but increases slightly when using higher call rates.

Interceptor overhead

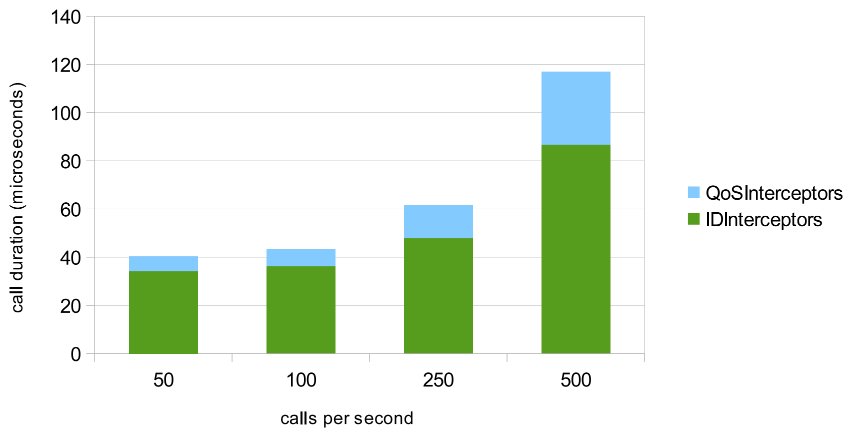


Figure 7.3: Direct Influence of the Interceptors

A part of the total overhead is caused by the used interceptors. As the interceptors are called serially and synchronously, their run time has a direct influence on the round trip time of a service call. To directly measure this overhead, the run time of the interceptors at different call rates was averaged over three minutes. Before measuring, a warm-up period of three minutes was used, as in the overall measurement. The QoSInterceptor share shows the time it takes to get the current time and add it to the measurement queue. IDInterceptor time is the time it takes to add `call-` and `method-IDs` to the message and replace old ones if necessary. The measurement shows that the `IDInterceptor` generally takes more time than the `QoSInterceptor`. When comparing percentages, the `QoSInterceptor` starts at 15,2%, at 500 calls per second the share grows to 25,8%.

7.1.3 Performance of the Monitor

It is more difficult to measure the performance of the monitor, as the events are queued when they are added. Therefore the response time is not a useful indicator of performance. When looking at performance tests for Esper, events per second are used to compare performance on different computer systems. As first tests showed, causality added considerable memory overhead. To avoid memory overflows, causality was deactivated for these tests. Nevertheless, it does not influence the number of events generated. For a discussion on this, see Subsection 7.2.2.

To get an overview of the load the framework generates, the generated events and the events coming from the MLF are counted. [37] recommends the following parameters for the Sun Java VM: `-Xms2g -Xmx2g -XX:NewSize=128m -XX:MaxNewSize=128m`
This sets the initial and maximum heap size to 2 GB and the heap memory size for fresh objects to 128 MB. Service and client are configured to send at maximum 150 events in one try (`MAX_BATCH_SEND`) and to wait 100ms if idle (`SENDING_IDLE_DELAY`). To get the neces-

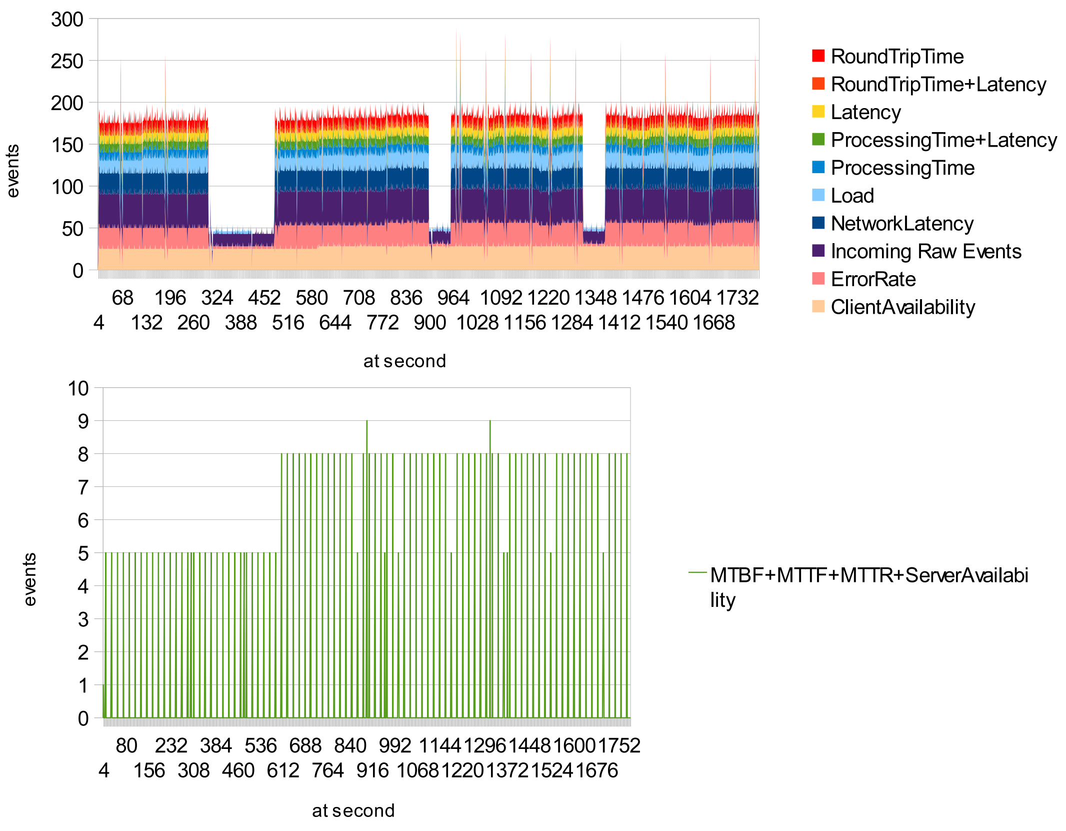


Figure 7.4: Event per 10 Seconds, Generated From the Framework

sary event numbers, Esper statement metrics were used. Esper statement metrics allow to output the number of incoming (input), and outgoing (insert and remove stream) events. In general, the activation of the performance metrics impacts performance, but as the interest only lies in obtaining the event counts, this is not relevant in our context. The number of transformations and MPs which generate events can be seen in Table 7.1.

To achieve the results from Figure 7.4, the performance metrics were queried every ten seconds. The statements were grouped by metric and the event count summed up. Some statements are shared by multiple metrics (e.g., by RoundTripTime and Latency). The first diagram in the figure shows the metrics which generate a lot of events regularly. Visible peaks which follow gaps are caused by delayed calls at the borders of the query interval. The second diagram shows the rest of the metrics, which generate few events over a longer timespan. In addition to the events generated by the transformations, every constraint fires an event if one of its metric values changes. If it is violated, it fires events, depending on the repetition type (see Subsection 5.3.2). For every `ConstraintEvent` that is fired, an additional `ActionEvent` is created.

7.2 Qualitative Evaluation, Open Problems and Limitations

This section contains a critical evaluation of the results of this thesis. It discusses different aspects, like the impact on the system and performance, the expressiveness of the DSLs, security and trust. At the end, possible areas of application are mentioned.

7.2.1 Platform Impact

When using an additional framework for QoS monitoring, the impact of this decision has to be weighed. One goal of our system is to be easily integrated into existing applications. Another is to be as platform independent as possible.

Regarding the first point, there is not much need for change, when incorporating the measurement functionality into other systems. To use the current prototype, all that is needed is adding the feature to the Web service interface (see Section 6.1.2) and some minor configuration of the measurement system (see Section 6.1.2). An important thing to consider is, that there must be an MLF for every supported Web service framework. The system is independently configurable and uses an additional and configurable interface for MP management. Nonetheless, two things have to be considered when using the framework. First, it sends a call ID with the message. This adds a bit of overhead to the SOAP message, concerning the size, but has no influence on the rest of the message content, as it is send through SOAP headers. Second, it influences the performance of the measured system by using system resources and by adding the call ID to the SOAP message.

Concerning the platform independence, the monitor itself must not be platform independent as it is decoupled from the MLF, which resides at the clients and services. In the current prototype, the MLF uses Java RMI for communication with the monitor. A first Web service-based solution proved to be very slow and inefficient. In future versions, this should be changed so a more system independent method, like sockets. The MLF implements measurement functionality for a certain system and can therefore not be totally system independent. This highlights a design challenge for MPs. It is possible to use the MLFs as a layer for platform independence, but this requires very general MP definitions which then could be mapped to the currently used architecture by the MLF. This would reduce the granularity of where the MPs are "attached" and might not be wishful. However, if fine-grained MPs, like the demonstrated PMPs, are used, the metric is dependent on a specific MLF (in our case the MLF for Apache CXF). This is a problem especially for clients, where it can not always be said beforehand, which framework the client uses. The more fine grained the MP, the more platform dependent the metrics get. So, depending on the use case, different granularity MPs might be interesting.

7.2.2 Performance Impact

Another influence on the system is caused by the resource usage of the framework, which results in a slow down of the measured application and a skewing of the measurement results. Concerning performance, there are two relevant parts of the system which should be regarded separately.

The MLF is the first part. As already mentioned in the quantitative evaluation, it has a direct influence on the calling process. The evaluation showed that the performance of the framework is not very big, especially concerning the measurement and the ID handling. ID handling, which is necessary for correlation of the measurements, has an additional influence of the measurements, as a call ID has to be sent with the message. This is, however, a lot less than sending measurement results with the message and allows to calculate metrics which are combined from client and server measurements. As the sending of the measurement is done in a separate Thread, it has no direct influence, but as the number of events can be very high (depending on the MDs), the sending overhead must not be ignored. Therefore it is very important to keep the sending overhead small by choosing an appropriate sending configuration (`MAX_BATCH_SEND` and `SENDING_IDLE_DELAY`) and by keeping the monitor close to the MLF. Keeping the monitor close to the MLF is not always possible for clients, but clients generally have much less calls per timespan than services. A big performance improvement came from substituting Web service calls with Java RMI calls in the prototype. A socket-based solution, as it is also recommended in [115], would further improve the call times and the system independence.

The monitor has only indirect influence which is further reduced by enqueueing the measurements. If the monitor blocks longer, because of an overload, the resulting delay can, however, cause a accumulation of events in the MLF which increases its resource consumption.

One reason for possible performance problems at the monitor is the number of transformations and events. The generated events can be seen in Figure 7.4, while the number of generated transformations is listed in Table 7.1. Both these values, and the incoming events of each transformation have an impact on the performance of Esper. As the proposed system uses limited-functionality DSLs for the generation of MDs, most metrics could be expressed using less transformations and therefore generating less events. Although the Esper solution patterns [40] advise to split complex statements into simpler substatements and smaller statements can be reused by other metrics, it can be assumed that an EPL-only version would, in many cases, be more efficient. Often functions add statements which could be combined with other statements to improve performance. Aggregations with sliding windows lead to a rapid increase in events by causing two output events per input event. This is especially a problem when aggregating over many calls, as for example in `Latency`, `ErrorRate` or `ClientAvailability`. To decrease the sent events, tumbling windows could be used instead of sliding windows. Instead of adding two events per incoming event, this only adds one event per defined timespan. If sliding windows are required, another solution could be the introduction of new functions. The main thing to notice is, that metric designers must be well aware of the underlying event based system to write efficient metrics. For example, some of the current metrics (`MeanTimeBetweenFailures`, `MeanTimeToFailure` and `MeanTimeToRepair`) currently use client calls to determine if the service is available. This could be made more efficient, if the mechanisms for `ServerAvailability` would be used. `ClientAvailability` itself could be improved by using a tumbling average aggregation.

Another problem of the monitor is its memory consumption. A problem which occurred early in testing was the huge overhead of the causality mechanism. The system stored references to all events which were causing another events in the caused event. For big aggregations over

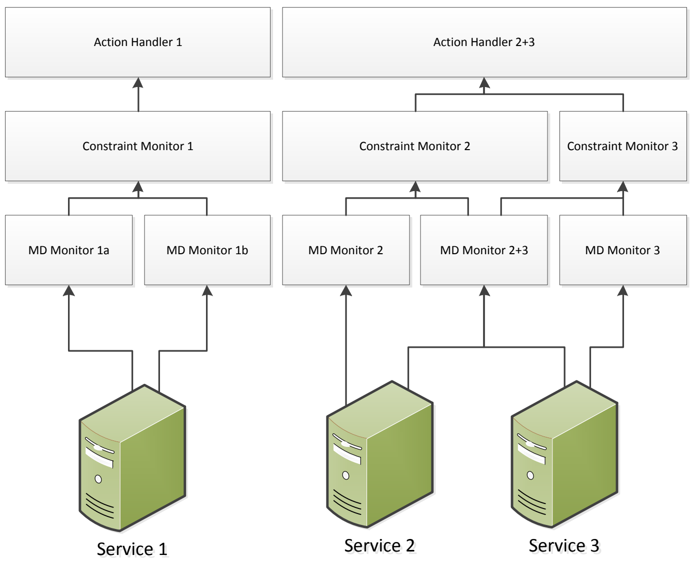


Figure 7.5: Potential Scaling of the Monitor.

lots of events, this caused the heap to overflow. To counter this, causality was deactivated for the MDDSL functions while testing. As a possible solution, causality could be made optional for all functions. This way, developers can select themselves if events should be retained for a function. In `ClientAvailability`, storing the events of thousands of calls could fill the heap, but averaging over the last 20 times to failures might be acceptable. A rule of thumb could be to keep aggregation windows short, and tumbling, if possible.

The general problem of the monitor requiring lots of events and memory could be handled by splitting the monitor into multiple instances. One instance could be responsible for one group of services, one service or even one metric. This instance should be close to the service, as this is where most events are generated. To add additional scalability, the computation of a single metric can also be split over more than one computer system. A potentially very useful separation could be introduced by separating the event system of the monitor according to different layers of events. The first layer is the transformation of the input events to metric values. As mentioned before, this could also be split by metrics and/or services. The next layer computes the constraints and determines which actions are to take. The last layer received action events and executes the corresponding action. An image of this structure can be found in Figure 7.5.

7.2.3 Expressiveness of the DSLs

The proposed solution uses three DSLs with different goals and different stakeholders. This subsection deals with the expressiveness of each of the languages and possible extensions.

MDDSL

The MDDSL provides the functions which are used to build the MDs, which are used to calculate the metrics. Based on a fixed set of MPs, the expressiveness of the MDDSL limits the possible supported metrics. Additional MPs could extend the functionality further, but this requires Java implementation effort. If there are no special reasons, MPs should be kept simple and calculations done using the MDDSL.

The functions of the MDDSL can be used for value comparisons, transformations of values, conditions, calculations and different aggregations. They are deliberately kept very general to support a wide rage of features, to keep the number of functions small and to foster reuse. To determine the necessary functions, interesting QoS attributes were collected and examined. This means that the functions are not a complete set. Also, application-specific metrics are very diverse, which makes the selection of appropriate general functions difficult. As the granularity of the functions is very coarse, often many functions are needed to express a certain metric. This creates many events which would not be needed with a more compact transformation. Through the use of Frag, it is however easily possible to add new functions.

The current framework already supports different types of measurements. Performance-based metrics are supported on a per-call, per-method or general level, availability-based metrics can be implemented and even basic application-specific metrics are supported. Application-specific metrics are difficult to handle, especially when they should be associated with a specific call, this is discussed in the MPDSL section. General metrics are also very diverse, but the development of the language showed that many attributes can be calculated using a small set of functions. If the MPs are flexible enough, it is often possible to add new types of metrics without adding functions or MPs. For example, "scalability" could be implemented as the relation of performance to load. Other metrics require additional MPs as sensors, but the rest of the calculation is very similar to other measurement (e.g., reliability - how good does the service fulfill the SLA).

Currently, most of the extensibility of the framework is achieved by adding new MPs. However, the MDDSL can also be extended by adding new functions to the DSL using the Frag programming language. When adding such a new functions, only a class and a `generateEsper`-method for the function has to be generated. This method must correspond to some rules and conventions. It should handle aggregation types (like `SAME_CALL` and `SAME_METHOD`), it must generate a name and it must generate a transformation which generates events with the name of the function and its parameters as the source. For performance and extensibility reasons it might also be interesting to add a function which can be used to dynamically add Esper statements inside of metrics. This would reduce the need to create additional functions in Frag.

A big limitation of the current MDDSL lies in type support. Currently only floating point measurements are supported. [113] mentions nominal and ordinal properties, which could also be implemented. Both types can be simulated by using fixed values, but real support would be more convenient. Textual properties are not supported at all. Measuring currently uses `long`-timestamps, calculations use `BigDecimal` values. Using `BigDecimal` instead of `double` for more precision added some problems. `BigDecimal` values can not be divided easily, as sometimes the result is not terminating in decimal notation, therefore a "division"-function had to be added for calculations. This function allows to specify a precision and a rounding method (for an example see the `Load` function in Section B.1.1), which are used if there is no terminating result . The Esper `AVG` aggregation supports `BigDecimal`, but in some cases the resulting average value can not be expressed in a decimal notation. Therefore a customized aggregation function had to be defined, which is used in the MDDSL `AVG` function. If the average is not expressible in decimal notation, it rounds the value according to the parameters.

Another future challenge is the initialisation of the metrics. This problem has two manifestations. First, when using aggregations, calculations, etc. any values which were measured before the system was turned off and on again, are lost. One effect of this is that any long running metrics, like `MeanTimeBetweenFailures`, do not work correctly if the system was shut down in between. The calculation function can fire only if all used values are present. While this seems obvious at first, if one needs to calculate a relation like `a/a+b` where `a` is the number of successful calls and `b` is the number of erroneous calls, this leads to problems. As long as only successful calls (`a`) are received, the calculation does not send out new events. To be able to compute this, `b` must be initialized with zero, but `a` must not be initialized, as this would result in a division by zero. For some functions it might be feasible to initialize some values with defaults, but some require historical data. This data could be acquired from a historic database, but this requires additional storage of low-level events.

Another interesting fact, caused by the lack of historic data for a certain measurement, can be seen at the beginning of the `ErrorRate` measurement in Figure 7.1. The `ErrorRate` is normally calculated over a timespan of five minutes. Directly after the start, this timespan is not yet available, leading to a much stronger influence of a single error to the total measurement. This could already trigger some actions. When changing a metric in the monitor, all modified statements (e.g., their aggregation times) are reset. Old aggregated values are no longer used in the new statement. This is caused by Esper's design and can not easily be changed. All statements which receive events from theses statements are also updated and reset, as the naming convention requires the statements to change their "source" parameter. At least limited initialization support should be implemented through special MDDSL constructs or function properties.

A limitation concerning time handling has already been mentioned in Section 5.3.3. The monitor can currently only aggregate over monitor time. A "manual" aggregation scheme which used MLF could be implemented, but this would have problems with events falling out of aggregation windows. If a periodic timestamp call was sent by the service, this could be used to advance time. This would, however, only be used for the services, and make the monitor more dependent on a trustable MLF.

An interesting extension for the metric definition would be the inclusion of a metric ontology. This could make our measurements better comparable for semantic based systems.

SLADSL

The current SLA is kept very simple to support the corresponding stakeholders. It supports constraints on single services and actions if those constraints are violated. For smaller applications, this might be acceptable, but there are some circumstances that require additional features of the SLA.

[107] proposes an ontology which supports different units and conversions between them. In our system, all metrics use a single unit. Inside the MD, the values are converted statically to this unit via event processing. The SLA must use the same unit when referencing the metric. This could be improved by using automatic unit conversions when interpreting the SLADSL. This would reduce the event count of most metrics and make editing the SLAs easier.

To make editing and metric generation more graphic, aggregation functions could be added to the SLA. This could allow to formulate constraints like “80% of the processingTime-values in the last 2 minutes must be smaller than 10ms” or similar. Also current default metrics which already calculate means or sums could be simplified (e.g., TimeToFailure instead of MeanTimeToFailre) and only aggregated in the SLA. When using the split monitor variant from Subsection 7.2.2, this could also shift some of the load to the lesser used constraint monitors, if wanted.

Parametrization of metrics could further improve the SLADSL. This would also impact the MDDSL, which then must support placeholders for parameters. Example parameters could include aggregation window sizes, maximum waiting times or even calculations. To support the measurement of specific methods of a services interface, it would also be required to add method name parameters to the metric. Otherwise, the MDs would be dependent on the service, which is discouraged. The `method-ID` is already present for PMPs, therefore a change would only require the addition of parameters to metrics.

Some works require an SLA to have additional attributes, e.g., a validity period [25]. This is not supported at the moment, but could be at future iterations. Another limitation is the support of multiple services but no exact specification of clients. With the current solution it is neither possible to define constraints which constrain a single general client, nor a single manually specified client. Support for the first could be added by adding a `SAME_CLIENT` aggregation type, which can be added in parallel to `SAME_CALL` or `SAME_METHOD`. The second could be supported by adding filters which filter by client address. Both solutions require a unique client identification which must not only consist of the IP address, but of an additional identifying token. Metric also can not be calculated over multiple services, caused by the structure of the SLA

Some interesting approaches can be seen in literature. Different “QoS Levels” are used to provide different QoS guarantees to different clients [105]. This is neither supported by the SLA nor by the framework itself. The current solution only monitors QoS and is not used for QoS management. To monitor such a setup, it would be required to implement client identification, as mentioned before. The second interesting approach is “Soft Contracts”. [90] uses probability distributions to detect possible future SLA violations. This could be used to fire warning actions before violations actually occur, therefore preventing punishments.

MPDSL

The MPDSL uses the Frag language to build an MP in form of a class, making it highly flexible. An MP only consists of parameters which have to be understood and interpreted by the MLF. Concerning MPs, the MLFs could be extended in a variety of ways.

As already mentioned before, the granularity of MPs is difficult to find. When using a flexible monitor, the MLFs could advertise their supported MPs and the monitor could choose which ones to activate, based on the measurement. The need for negotiating monitoring capabilities is mentioned in [20], although on another level of granularity. Conversions between different MPs are also an option.

A big bonus of using the MPDSL is the extensibility it supports. The following list of possible new or changed MPs is by far not complete. It should also be considered, that an MLF must not reside at the client or service, but all it needs is a interface for the MP distribution.

- Application-Specific MPs are currently very simple. It is not possible to associate an application-specific measurement with a single call, as the implementation of the MLF does not support this. This was a deliberate decision, as adding this would have had a much higher influence on the application as is was the goal of this thesis. For some uses, this might not be so important.
- PMPs could be extended to support other locations than services or clients like, for example, SOAP intermediaries. This could also be used to simulate proxy-based measurements, where appropriate. When monitoring compositions, an MLF could exist for the whole composition, and an extended PMP could be specified to attach to a location inside a used service. VRESCo, for example, uses TCP monitoring, which could also be added through additional MPs.
- A very interesting MP could be used to measure the adherence to an SLA. This can be used to measure a kind of reliability.
- Some MPs could support active polling of resources. That way, MPs combine the functionality of agents and probes from [115]. In addition, this would enable the framework to measure system which lie outside of its influence.
- For an even better extensibility, an MP in an MLF could act as an adapter for another measurement framework.
- To better support application-specific measurements, special MPs could be added which extract data from requests and/or responses via XPath. While this could lead to a more flexible QoS measurement and could also shift some measurement to the application-level if needed, the usage of XPath could also break streaming, as the whole SOAP message must be read to build up a document model.

The last two types of MPs would provide the most benefit, if the framework would support textual data as values.

7.2.4 Security and Trust

An important topic, especially for publicly available software is security. At the moment, the prototype does not support authentication. This could, however, be implemented easily. To be sure that a measurement comes from a certain service, each service could either be authenticated using a secret keyword or by using encryption. Both could be stored at the monitor with the service registration. For most clients this is of no use, as client are often not known before runtime. To make sure measurements can be tracked to a client, at least am IP address should be extracted from the request and handed down for some measurements by using causality. It might even be interesting to allow `SAME_CLIENT` aggregations in the MDDSL. For static clients, registration and automatic MP updates could be added. On the other side, services must be sure that MP updates are legitimate, so notifications from the monitor must also use authentication. In this case, a digital signature would be an appropriate means of authentication. For clients, who request MP updates, this would also provide additional security. Authentication should be provided at messaging framework level to avoid overhead from processing unauthenticated requests and therefore open doors for denial of service attacks.

The above measures can make sure that a measurement comes from the right source, but as the service providers generally have interest in manipulating measurements to their benefit, steps have to be taken to prevent fake measurements. One problem stems from the way time is managed in the system when using PMPs or RMPs. As already mentioned in Section 5.3.3, time handling is a challenge in a distributed system. When using the time at the MLF, the clock could possibly be subject to manipulations by the service provider, event though the Java `System.nanoTime()` is used instead of `System.currentTimeMillis()`. Another possibility to manipulate the time is the editing of messages which are sent to the monitor. At the moment, the client generates call IDs. The monitor can not be sure that the client does not use two identical call IDs in short succession, which would disturb the event calculation.

To solve these challenges, there are two complementary solutions. First, the monitor could verify the events. As the monitor is agnostic of the measured frameworks, the knowledge must come from the metric designers. Some verifications might be added to the DSL, or additional EPL statements could be added to check for manipulation attempts (e.g., duplicate phases for one call ID, backward running time, etc.). Another possibility for monitoring services could be to use regularly sent events to check for discrepancies in local and remote time. [57] proposes the use of client feedback to verify measurements at the monitor. The other solution is provided by using a trusted module. This could be difficult, as the measurement itself is done at the clients/services CXF framework. If would be required to check for the identity of the attached MPs regularly and to implement the MLF in a tamper-proof way. This could provide almost impossible if installed on a foreign computer system.

7.2.5 Areas of Application

Based on the evaluation above, there are two relevant areas of applications. First, it could be used for the testing of a service prior to publishing. At this constellation, some of the challenges can be easily cast aside. Especially security and trust can be neglected. On the other side, the metric distribution system would only be useful if the measured system is very complex.

Another possibility is to use it on a running productive system. While the prototype is not yet suitable for any use in a productive environment, future versions could be used to monitor such systems. In that case, all of the above mentioned challenges, as well as benefits apply. Special care must be put on checking trustability, platform-independence (especially for the clients) and performance. A benefit of this system is, that it can easily be deactivated at runtime without any negative effects. If no metrics are used, the only influence on the measured system is the used memory and the existance of the `Distribution Interface` (only at services).

8. Conclusion and Future Work

The SOA paradigm gets more and more widespread and many companies decide to use services for their infrastructure. Big corporations like Amazon [4] provide additional Web service for different purposes. This leads to a wide availability of services. The opportunity to use a variety of foreign services directly or via compositions makes it more important to monitor the service quality. The contracts are defined using SLAs. In literature, there are lots of solutions for service monitoring and QoS managing, but many lack a detailed specification of how to measure specific QoS metrics.

One of the goals of this thesis was to make the metric specification more detailed, so that the exact measuring description can be compared. The second research goals was to find a new method how we can monitor many QoS properties of a service with as less impact on the service as possible. The final goal was to find out how these measurements can be collected and analyzed. The solution should have a minimal performance footprint and should have as less influence on the service implementation as possible. It should also be able to measure a broad spectrum of QoS values and remaining extensible for new, yet unspecified, measurements.

To achieve this, several DSLs were used. Each has a different purpose and is used at a different time. The SLADSL is used for SLA description, the MPDSL specifies measuring points at the measurement location and the MDDSL is used to describe metrics, based on the MPs defined in the MPDSLs. Metric descriptions are collected at a centralized monitor and are used in an SLA. The used measuring points are distributed through a distribution system to the corresponding measurement locations. The measurements are sent from the locations directly to the monitor, where they are processed, using a CEPs framework. Potential violations of the SLA are then detected and appropriate actions, which were specified in the SLA, taken.

This thesis showed that DSLs are well suited for the purpose of describing SLAs and how to measure certain metrics. As application-specific QoS properties are very diverse, the MDDSL should be extended to allow for more support of application-specific metrics. A literature review shows, that the currently used SLA also has room for improvement by adding more functionality.

Performance impact on the measurement location is negligibly small, whereas the central monitor proves to be a potential bottleneck in the scenario. Nevertheless, the event-based structure of the monitor makes it easy to split it up, providing good scalability, from the service to the metric level.

In total, it can be said, that the measurement of QoS and the description of SLAs is of a very wide scope. Therefore many additional features can still be added to form a complete measurement solution. The flexibility and extensibility of the framework, especially the use of DSLs and the event-based structure of the monitor, facilitate the adding of new functionality and make this solution very useful.

8.1 Future Work

As already said, the architecture of the system leaves a lot of room for extensions. An interesting step would be to test the system under circumstances, which are closer to the real world, like using more, distributed, clients and therefore a more realistic load. This could lead the way to better performance optimizations.

To make the system better usable in practice, the centralized monitor should be split up into a set of smaller monitors, as proposed in Figure 7.5. Doing this requires some modifications of the monitor code, but the structure allows for an easy separation. This would improve the monitors performance and remove the monitor as a bottleneck. Another important step would be to add authentication to the MLFs and the monitor, to prevent attacks from external sources, and better support for trustability. One thing that is not yet implemented in the prototype is the implementation of *optional* causality. As the evaluation showed, the proposed method of adding causality leads to an extensive use of memory in some aggregations. Making causality optional for every metric could keep both advantages, the lower memory profile and the keeping of causality, where necessary. To further improve the prototype, sockets should be used instead of Java RMI, as this would increase performance and platform-independence.

To improve the area of use, the SLADSLs should be extended to support additional features like a validity period, unit conversions, aggregate functions and potentially soft contracts. If the MDDSL is updated to support parameters for MDs, the MDDSL can better support client-side constraints and make SLAs more expressive. More functions for MDs would not only allow to define a broader spectrum of application-specific QoS metrics, but might also improve performance by reducing the number of used functions for each metric. An improvement of the usability of the system could be achieved by providing a detailed instruction of how to add new functions to the MDDSLs and a guide how to design MPs.

A. List of Abbreviations

AOP Aspect-oriented Programming

AMP Application-Specific Measuring Point

AtMP Application-Specific Time Measuring Point

AvMP Application-Specific Value Measuring Point

API Application Programming Interface

BPEL Business Process Execution Language

CEP Complex Event Processing

DBMS Database Management System

DSL domain-specific language

DSEL domain-specific embedded language

EBNF Extended Backus-Naur Form

ECA Event-Condition-Action

EPL Event Processing Language

ESP Event Stream Processing

GPL general purpose language

HTTP Hypertext Transfer Protocol

JMS Java Message Service

JMX Java Management Extensions

JPA Java Persistence API

JSF JavaServer Faces

MIME Multipurpose Internet Mail Extensions

MD Measurement Description

MDDSL Measurement Description DSL

MLF	Measuring Location Framework
MP	Measuring Point
MPDSL	Measuring Point DSL
MTBF	Mean Time Between Failures
MTTR	Mean Time To Repair
NTP	Network Time Protocol
OASIS	Organization for the Advancement of Structured Information Standards
OCL	Object Constraint Language
PMP	Phase Measuring Point
POSET	partially ordered set
POJO	Plain Old Java Object
QoS	Quality of Service
QuaLa	Quality of Service Language
RPC	remote procedure call
RMI	remote method invocation
RMP	Repeated Measuring Point
RTML	Run-Time Monitor Specification Language
SLA	Services Level Agreement
SLADSL	Services Level Agreement DSL
SLO	Services Level Objective
SOA	Services-Oriented Architecture
SOC	Service-Oriented Computing
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
UDDI	Universal Description, Discovery and Integration
UI	user interface
URI	Uniform Resource Identifier

URL Uniform Resource Locator

UML Unified Modeling Language

VRESCo Vienna Runtime Environment for Service-oriented Computing

WS Web service

WSCol Web Service Constraint Language

WSDL Web Services Description Language

WSLA Web Service Level Agreements

WSML Web Service Modeling Language

WSMO Web Service Modeling Ontology

WSOL Web Service Offerings Language

WS-CDL Web Services Choreography Description Language

W3C World Wide Web Consortium

XML Extensible Markup Language

B. Listings

B.1 Used Metrics

B.1.1 Standard Metrics

Network Latency

```
## NETWORK LATENCY ##
set concern [desc::Metric create -name "NetworkLatency"]
  set milliLatency [desc::CALCULATION create]
    set totalSendTime [desc::TIMEBETWEEN create]
        set fromPoint [cxf::PhaseMeasuringPoint create]
          $fromPoint phase ClientOutSend
        set toPoint [cxf::PhaseMeasuringPoint create]
          $toPoint phase ClientInReceive
    $totalSendTime from $fromPoint
    $totalSendTime to $toPoint
    $totalSendTime aggregation SAME_CALL
    $totalSendTime maximumWaitInterval [desc::TimeInterval create -value 2 -unit MINUTES]
    set serverRoundTrip [desc::TIMEBETWEEN create]
        set fromPoint [cxf::PhaseMeasuringPoint create]
          $fromPoint phase ServerInReceive
        set toPoint [desc::UNION create]
          set faultPoint [cxf::PhaseMeasuringPoint create]
            $faultPoint phase ServerFaultOutSend
          set okPoint [cxf::PhaseMeasuringPoint create]
            $okPoint phase ServerOutSend
        $toPoint events [list build $faultPoint $okPoint]
    $serverRoundTrip from $fromPoint
    $serverRoundTrip to $toPoint
    $serverRoundTrip aggregation SAME_CALL
    $serverRoundTrip maximumWaitInterval [desc::TimeInterval create -value 2 -unit MINUTES]
  $milliLatency calculation `(TST-SRT)/1000000`
  $milliLatency parameters [Hashtable create -set TST $totalSendTime -set SRT $serverRoundTrip]
  $milliLatency aggregation SAME_CALL
  $milliLatency maximumWaitInterval [desc::TimeInterval create -value 1 -unit MINUTES]
$concern value $milliLatency
$concern unit MILLISECONDS
```

Latency

```
## LATENCY ##
set concern [desc::Metric create -name "Latency"]
  set milliLatency [desc::CALCULATION create]
    set roundTripTime [desc::TIMEBETWEEN create]
        set fromPoint [cxf::PhaseMeasuringPoint create]
        $fromPoint phase ClientOutSetup
        set toPoint [cxf::PhaseMeasuringPoint create]
        $toPoint phase ClientOutSetupEnding
    $roundTripTime from $fromPoint
    $roundTripTime to $toPoint
    $roundTripTime aggregation SAME_CALL
    $roundTripTime maximumWaitInterval [desc::TimeInterval create -value 2
        -unit MINUTES]
    set processingTime [desc::TIMEBETWEEN create]
        set fromPoint [cxf::PhaseMeasuringPoint create]
          $fromPoint phase ServerInInvoke
          $fromPoint before [list build
              `org.apache.cxf.interceptor.ServiceInvokerInterceptor`]
        set toPoint [desc::UNION create]
          set faultPoint [cxf::PhaseMeasuringPoint create]
          $faultPoint phase ServerInInvoke
          $faultPoint before [list build
              `org.apache.cxf.interceptor.ServiceInvokerInterceptor`]
          $faultPoint isFault true
          set okPoint [cxf::PhaseMeasuringPoint create]
          $okPoint phase ServerInInvoke
          $okPoint after [list build
              `org.apache.cxf.interceptor.ServiceInvokerInterceptor`]
        $toPoint events [list build $faultPoint $okPoint]
    $processingTime from $fromPoint
    $processingTime to $toPoint
    $processingTime aggregation SAME_CALL
    $processingTime maximumWaitInterval [desc::TimeInterval create -value 2
        -unit MINUTES]
  $milliLatency calculation `(RTT-PT)/1000000`
  $milliLatency parameters [Hashtable create -set RTT $roundTripTime -set PT
      $processingTime]
  $milliLatency aggregation SAME_CALL
  $milliLatency maximumWaitInterval [desc::TimeInterval create -value 1
      -unit MINUTES]
$concern value $milliLatency
$concern unit MILLISECONDS
```

Client Availability

```
## CLIENT AVAILABILITY ##
set concern [desc::Metric create -name "ClientAvailability"]
  set relationInPercent [desc::CALCULATION create]
    set relation [desc::AVG create]
      set okOrFault [desc::VALUEUNION create]
        set onOK [desc::SETVALUEONEVENT create]
          set okMP [cxf::PhaseMeasuringPoint create]
          $okMP phase ClientOutSetupEnding
        $onOK on $okMP
        $onOK value 1
        set onERR [desc::SETVALUEONEVENT create]
          set faultMP [cxf::PhaseMeasuringPoint create]
          $faultMP phase ClientOutSetup
          $faultMP isFault true
        $onERR on $faultMP
        $onERR value 0
      $okOrFault events [list build $onOK $onERR]
    $relation of $okOrFault
    $relation interval [desc::AggregationInterval create -value 10 -unit
        MINUTES]
  $relationInPercent calculation `REL*100`
  $relationInPercent parameters [Hashtable create -set REL $relation]
$concern value $relationInPercent
$concern unit PERCENT
```

Load

```
## LOAD ##
set concern [desc::Metric create -name "Load"]
  set countPerSecond [desc::CALCULATION create]
    set countEvents [desc::COUNT create]
      set receiveMP [cxf::PhaseMeasuringPoint create]
      $receiveMP phase ServerInReceive
    $countEvents of $receiveMP
    $countEvents interval [desc::TimeInterval create -value 2 -unit MINUTES]
  $countPerSecond calculation
      `division(CNT, cast(2*60,BigDecimal),5,RoundingMode.HALF_UP)`
  $countPerSecond parameters [Hashtable create -set CNT $countEvents]
$concern value $countPerSecond
$concern unit TIMES
```

Mean Time Between Failures (MTBF)

```
## MEAN TIME BETWEEN FAILURES ##
set concern [desc::Metric create -name "MeanTimeBetweenFailures"]
  set milliMTBF [desc::CALCULATION create]
    set avgTimeBetwFail [desc::AVG create]
      set timeBetweenFailures [desc::INTERVAL create]
        set onFault [desc::ONEVENTCHANGE create]
          set okMP [cxf::PhaseMeasuringPoint create]
            $okMP phase ClientOutSetupEnding
          set faultMP [cxf::PhaseMeasuringPoint create]
            $faultMP phase ClientOutSetup
            $faultMP isFault true
        $onFault events [list build $okMP $faultMP]
        $onFault fireOn [list build $faultMP]
      $timeBetweenFailures between $onFault
    $avgTimeBetwFail of $timeBetweenFailures
    $avgTimeBetwFail interval [desc::AggregationInterval create -value 20
        -unit EVENTS]##or TIME?
  $milliMTBF calculation `MTBF/1000000`
  $milliMTBF parameters [Hashtable create -set MTBF $avgTimeBetwFail]
$concern value $milliMTBF
$concern unit MILLISECONDS
```

Mean Time To Failure (MTTF)

```
## MEAN TIME TO FAILURE ##
set concern [desc::Metric create -name "MeanTimeToFailure"]
  set milliMTTF [desc::CALCULATION create]
    set avgTimeToFail [desc::AVG create]
      set timeToFail [desc::TIMEBETWEEN create]
          set okMP [cxf::PhaseMeasuringPoint create]
          $okMP phase ClientOutSetupEnding
          set faultMP [cxf::PhaseMeasuringPoint create]
          $faultMP phase ClientOutSetup
          $faultMP isFault true
        set onFault [desc::ONEVENTCHANGE create]
        $onFault events [list build $okMP $faultMP]
        $onFault fireOn [list build $faultMP]
        set onRepair [desc::ONEVENTCHANGE create]
        $onRepair events [list build $okMP $faultMP]
        $onRepair fireOn [list build $okMP]
      $timeToFail from $onRepair
      $timeToFail to $onFault
      $timeToFail maximumWaitInterval [desc::TimeInterval create -value 7
          -unit DAYS]
    $avgTimeToFail of $timeToFail
    $avgTimeToFail interval [desc::AggregationInterval create -value 20
        -unit EVENTS]##or TIME?
  $milliMTTF calculation `MTTF/1000000`
  $milliMTTF parameters [Hashtable create -set MTTF $avgTimeToFail]
$concern value $milliMTTF
$concern unit MILLISECONDS
```

Mean Time To Repair (MTTR)

```
## MEAN TIME TO REPAIR ##
set concern [desc::Metric create -name "MeanTimeToRepair"]
  set milliMTTR [desc::CALCULATION create]
    set avgTimeToRep [desc::AVG create]
      set timeToRepair [desc::TIMEBETWEEN create]
          set okMP [cxf::PhaseMeasuringPoint create]
          $okMP phase ClientOutSetupEnding
          set faultMP [cxf::PhaseMeasuringPoint create]
          $faultMP phase ClientOutSetup
          $faultMP isFault true
        set onFault [desc::ONEVENTCHANGE create]
        $onFault events [list build $okMP $faultMP]
        $onFault fireOn [list build $faultMP]
        set onRepair [desc::ONEVENTCHANGE create]
        $onRepair events [list build $okMP $faultMP]
        $onRepair fireOn [list build $okMP]
      $timeToRepair from $onFault
      $timeToRepair to $onRepair
      $timeToRepair maximumWaitInterval [desc::TimeInterval create -value 7
          -unit DAYS]
    $avgTimeToRep of $timeToRepair
    $avgTimeToRep interval [desc::AggregationInterval create -value 20 -unit
        EVENTS]##or TIME?
  $milliMTTR calculation ‘MTTR/1000000‘
  $milliMTTR parameters [Hashtable create -set MTTR $avgTimeToRep]
$concern value $milliMTTR
$concern unit MILLISECONDS
```

Round-Trip Time

```
## ROUND TRIP TIME ##
set concern [desc::Metric create -name "RoundTripTime"]
  set milliRTT [desc::CALCULATION create]
    set roundTripTime [desc::TIMEBETWEEN create]
        set fromPoint [cxf::PhaseMeasuringPoint create]
          $fromPoint phase ClientOutSetup
        set toPoint [cxf::PhaseMeasuringPoint create]
          $toPoint phase ClientOutSetupEnding
    $roundTripTime from $fromPoint
    $roundTripTime to $toPoint
    $roundTripTime aggregation SAME_CALL
    $roundTripTime maximumWaitInterval [desc::TimeInterval create -value 2
        -unit MINUTES]
  $milliRTT calculation ‘RTT/1000000‘
  $milliRTT parameters [Hashtable create -set RTT $roundTripTime]
$concern value $milliRTT
$concern unit MILLISECONDS
```

Server Availability

```
## SERVER AVAILABILITY ##
set concern [desc::Metric create -name "ServerAvailability"]
  set availabilityInPercent [desc::CALCULATION create]
    set averageAvailability [desc::AVG create]
      set isAvailable [desc::IF create]
        set tumblingCount [desc::TUMBLING_COUNT create]
          set repMP [cxf::RepeatedMeasuringPoint create]
            $repMP location SERVER
            $repMP interval [desc::TimeInterval create -value 10 -unit
                SECONDS]
        $tumblingCount of $repMP
        $tumblingCount interval [desc::TimeInterval create -value 20 -unit
            SECONDS]
      $isAvailable value $tumblingCount
      $isAvailable comparison "value>0"
      $isAvailable then 1
      $isAvailable else 0
    $averageAvailability of $isAvailable
    $averageAvailability interval [desc::AggregationInterval create -value
        10 -unit MINUTES]
  $availabilityInPercent calculation 'SA*100'
  $availabilityInPercent parameters [Hashtable create -set SA
      $averageAvailability]
$concern value $availabilityInPercent
$concern unit PERCENT
```

ErrorRate

```
## ERROR RATE ##
set concern [desc::Metric create -name "ErrorRate"]
  set relationInPercent [desc::CALCULATION create]
    set relation [desc::AVG create]
      set okOrExc [desc::VALUEUNION create]
        set onEXC [desc::SETVALUEONEVENT create]
          set exceptionMP [cxf::PhaseMeasuringPoint create]
          $exceptionMP phase ServerFaultOutSend
        $onEXC on $exceptionMP
        $onEXC value 1
        set onOK [desc::SETVALUEONEVENT create]
          set okMP [cxf::PhaseMeasuringPoint create]
          $okMP phase ServerOutSend
        $onOK on $okMP
        $onOK value 0
      $okOrExc events  [list build $onOK $onEXC]
    $relation of $okOrExc
    $relation interval [desc::AggregationInterval create -value 5 -unit
        MINUTES]
  $relationInPercent calculation 'REL*100'
  $relationInPercent parameters [Hashtable create -set REL $relation]
$concern value $relationInPercent
$concern unit PERCENT
```

Processing Time

```
## PROCESSING TIME ##
set concern [desc::Metric create -name "ProcessingTime"]
  set milliPT [desc::CALCULATION create]
    set processingTime [desc::TIMEBETWEEN create]
        set fromPoint [cxf::PhaseMeasuringPoint create]
          $fromPoint phase ServerInInvoke
          $fromPoint before [list build
              `org.apache.cxf.interceptor.ServiceInvokerInterceptor`]
        set toPoint [desc::UNION create]
          set faultPoint [cxf::PhaseMeasuringPoint create]
            $faultPoint phase ServerInInvoke
            $faultPoint before [list build
                `org.apache.cxf.interceptor.ServiceInvokerInterceptor`]
            $faultPoint isFault true
          set okPoint [cxf::PhaseMeasuringPoint create]
            $okPoint phase ServerInInvoke
            $okPoint after [list build
                `org.apache.cxf.interceptor.ServiceInvokerInterceptor`]
        $toPoint events [list build $faultPoint $okPoint]
    $processingTime from $fromPoint
    $processingTime to $toPoint
    $processingTime aggregation SAME_CALL
    $processingTime maximumWaitInterval [desc::TimeInterval create -value 2
        -unit MINUTES]
  $milliPT calculation `PT/1000000`
  $milliPT parameters [Hashtable create -set PT $processingTime]
$concern value $milliPT
$concern unit MILLISECONDS
```

B.1.2 Application-Specific Metrics

Search Result Accuracy

```
## SEARCH RESULT ACCURACY ##
set concern [desc::Metric create -name "SearchResultAccuracy"]
  set limitedAccuracy [desc::LIMIT create]
    set accuracyInPercent [desc::CALCULATION create]
      set countOfFinding [desc::COUNT create]
        set findingMP [cxf::AppSpecTimeMeasuringPoint create]
          $findingMP location CLIENT
          $findingMP customName `ResultOkay`
      $countOfFinding of $findingMP
      $countOfFinding interval [desc::TimeInterval create -value 5 -unit
          MINUTES]
      set countOfSearch [desc::COUNT create]
        set searchMP [cxf::AppSpecTimeMeasuringPoint create]
          $searchMP location SERVER
          $searchMP customName `SearchDone`
      $countOfSearch of $searchMP
      $countOfSearch interval [desc::TimeInterval create -value 5 -unit
          MINUTES]
    $accuracyInPercent calculation
        `division(100*FIND,SEARCH,5,RoundingMode.HALF_UP)`
    $accuracyInPercent parameters [Hashtable create -set FIND
        $countOfFinding -set SEARCH $countOfSearch]
  $limitedAccuracy of $accuracyInPercent
  $limitedAccuracy upper 100
  $limitedAccuracy upperInclusive true
  $limitedAccuracy lower 0
  $limitedAccuracy lowerInclusive true
$concern value $limitedAccuracy
$concern unit PERCENT
```

Proposition Quality

```
## PROPOSITION QUALITY ##
set concern [desc::Metric create -name "PropositionQuality"]
  set averageQuality [desc::AVG create]
    set limitedQuality [desc::LIMIT create]
      set propQuality [cxf::AppSpecValueMeasuringPoint create]
        $propQuality location CLIENT
        $propQuality customName "PropositionQuality"
    $limitedQuality of $propQuality
    $limitedQuality upper 100
    $limitedQuality upperInclusive true
    $limitedQuality lower 0
    $limitedQuality lowerInclusive true
  $averageQuality of $limitedQuality
  $averageQuality interval [desc::AggregationInterval create -value 10 -unit
      MINUTES]
$concern value $averageQuality
$concern unit PERCENT
```

B.2 Used SLA

```
WatchmeSLA
{
  Service
  {
    WHEN NOT ServerAvailability>=0% => mailto "matthias@happens.at",
    WHEN NOT ClientAvailability>=0% => mailto "matthias@happens.at",
    WHEN NOT (Latency>=0ms OR NetworkLatency>=0ms) AND RoundTripTime>=0ms =>
        mailto "matthias@happens.at",
    WHEN NOT ProcessingTime>=0ms => mailto "matthias@happens.at",
    WHEN NOT Load>=0times AND ErrorRate>=0% => mailto "matthias@happens.at",
    WHEN NOT MeanTimeBetweenFailures>=0ms AND (MeanTimeToRepair>=0ms OR
        MeanTimeToFailure>=0ms) => mailto "matthias@happens.at",
    WHEN NOT SearchResultAccuracy>=0% AND PropositionQuality>=0% => mailto
        "matthias@happens.at"
  }
}
```

B.3 Used MP Definitions

B.3.1 RMP

```
FMF::Enum create LocationEnum -setEnumValues {
  CLIENT SERVER
}
FMF::Class create RepeatedMeasuringPoint -superclasses
    desc::TimeMeasuringPoint -attributes {
  location LocationEnum
}
FMF::Association create RepeatedMeasuringPointInterval -ends {
  {RepeatedMeasuringPoint -roleName measuringPoint -navigable false
      -multiplicity *}
  {desc::TimeInterval -roleName interval -navigable true -multiplicity 1 }
}

RepeatedMeasuringPoint method getMPdata {service} {
  set mp [Object create]
  $mp set name [string build 'RMP['$service'|'[self location]'|'[[self
      interval] value][[self interval] unit]']']
  $mp set type "RMP"
  $mp set location [self location]

  $mp set interval [string build [[self interval] value][[self interval]
      unit]]
  return $mp
}
```

B.3.2 AMP

```
FMF::Class create AppSpecValueMeasuringPoint -superclasses desc::ValueMeasuringPoint -attributes {
  customName string
  location LocationEnum
}
FMF::Class create AppSpecTimeMeasuringPoint -superclasses desc::TimeMeasuringPoint -attributes {
  customName string
  location LocationEnum
}
AppSpecValueMeasuringPoint method getMPdata {service} {
  set mp [Object create]
  $mp set name [string build ‘AvMP[‘$service‘|‘[self location]‘|‘[self customName]‘]‘]
  $mp set type "AvMP"
  $mp set location [self location]

  $mp set customName [self customName]
  return $mp
}
AppSpecTimeMeasuringPoint method getMPdata {service} {
  set mp [Object create]
  $mp set name [string build ‘AtMP[‘$service‘|‘[self location]‘|‘[self customName]‘]‘]
  $mp set type "AtMP"
  $mp set location [self location]

  $mp set customName [self customName]
  return $mp
}
```

B.3.3 PMP

```
FMF::Enum create ChainTypeEnum -setEnumValues {
  CLIENT_IN CLIENT_OUT CLIENT_FAULT_IN SERVER_IN SERVER_OUT SERVER_FAULT_OUT
}
FMF::Class create Chain -attributes {
  type ChainTypeEnum
  location LocationEnum
}
FMF::Class create Phase -attributes {
  cxfname String
}
FMF::Aggregation create ChainPhases -ends {
  {Chain -roleName chain -multiplicity 1 -aggregatingEnd true -navigable true}
  {Phase -roleName phases -multiplicity * -navigable true}
}
FMF::Class create PhaseMeasuringPoint -superclasses desc::TimeMeasuringPoint -attributes {
   isFault boolean
```

```
    after string
    before string
  } -defaults {
    isFault false
  } ##after and before are lists
FMF::Association create PhaseMeasuringPointPhase -ends {
  {PhaseMeasuringPoint -roleName measuringPoint -navigable true
     -multiplicity *}
  {Phase -roleName phase -navigable true -multiplicity 1 }
}
FMF::Association create PhaseMeasuringPointChain -ends {
  {PhaseMeasuringPoint -roleName measuringPoint -navigable true
     -multiplicity *}
  {Chain -roleName chain -navigable true -multiplicity 1 }
}

PhaseMeasuringPoint method getMPdata {service} {
  if {[self isFault]} {set faultName 'FAULT'} else {set faultName 'NORMAL'}
  set afterList [list sort [self after]]
  set beforeList [list sort [self before]]
  set afterName [list join $afterList ","]
  set beforeName [list join $beforeList ","]

  set mp [Object create]
  $mp set name [string build 'PMP['$service'|'[[self phase] chain]'|'[[self
     phase] cxfname]'|'$faultName'|AFT|'$afterName'|BEF|'$beforeName']']
  $mp set type "PMP"
  $mp set location [[[self phase] chain] location]

  $mp set phase [[self phase] cxfname]
  $mp set chain [[self phase] chain]
  $mp set isFault [self isFault]
  $mp set afterList $afterName
  $mp set beforeList $beforeName
  return $mp
}

cxf::Chain create ClientIn -type CLIENT_IN -location CLIENT
cxf::Chain create ClientFaultIn -type CLIENT_FAULT_IN -location CLIENT
cxf::Chain create ServerIn -type SERVER_IN -location SERVER
cxf::Chain create ClientOut -type CLIENT_OUT -location CLIENT
cxf::Chain create ServerOut -type SERVER_OUT -location SERVER
cxf::Chain create ServerFaultOut -type SERVER_FAULT_OUT -location SERVER

  cxf::Phase create ClientInReceive -cxfname "receive" -chain ClientIn
  cxf::Phase create ClientInPreStream -cxfname "pre-stream" -chain ClientIn
  cxf::Phase create ClientInUserStream -cxfname "user-stream" -chain ClientIn
  cxf::Phase create ClientInPostStream -cxfname "post-stream" -chain ClientIn
  cxf::Phase create ClientInRead -cxfname "read" -chain ClientIn
  #[...]
```

B.4 Other listings

B.4.1 Default Velocity Template For Actions

```
#if($event.violated)
Service $event.serviceURI violated condition
    "$event.causingConstraintEvent.condition" when
    ${eventevent.causingConstraintEvent.valueMap}. Sending a message to
    $event.actionParameter via $event.actionName.
Caused By ConstraintEvent: ${event.causingConstraintEvent}
Caused By SLAEvents:
#foreach($slaCause in $event.causingConstraintEvent.causingSLAEvents)
  x $slaCause
    caused by:
#set( $currentQoSEvent = $slaCause.causingQoSValueEvent)
    #parse("templates/QoSEventCausesRecursive.tpl")

#end
#{else}
Service $event.serviceURI does not violate condition
    "$event.causingConstraintEvent.condition" when
    ${event.causingConstraintEvent.valueMap} anymore. Sending a message to
    $event.actionParameter via $event.actionName.
#end
```

Bibliography

[1] Phil Adams, Peter Easton, Eric Johnson, Roland Merrick, and Mark Phillips. SOAP over Java Message Service 1.0. W3C Working Draft, W3C, October 2010. `http://www.w3.org/TR/2010/WD-soapjms-20101026/`.

[2] M. Aida, N. Miyoshi, and K. Ishibashi. A scalable and lightweight QoS monitoring technique combining passive and active approaches. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1 of *INFOCOM 2003*, pages 125 – 133 vol.1, march-3 april 2003.

[3] Alves, Alexandre and Arkin, Assaf and Askary, Sid and Barreto, Charlton and Bloch, Ben and Curbera, Francisco and Ford, Mark and Goland, Yaron and Guízar, Alejandro and Kartha, Neelakantan and Kevin Liu, Canyang and Khalaf, Rania and König, Dieter and Marin, Mike and Mehta, Vinkesh and Thatte, Satish and van der Rijn, Danny and Yendluri, Prasad and Yiu, Alex. Web Services Business Process Execution Language Version 2.0. OASIS Standard, OASIS, April 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[4] Amazon Web Services. `http://aws.amazon.com/de/`. Last visited 10/31/11.

[5] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Open Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) WG, March 2007.

[6] Apache CXF. `http://cxf.apache.org/`. Last visited 06/20/11.

[7] Apache CXF - User's Guide. `http://cxf.apache.org/docs/index.html`. Last visited 06/02/11.

[8] The Apache Velocity Project. `http://velocity.apache.org/`. Last visited 08/26/11.

[9] Saeed Araban and Leon Sterling. Quality of Service for Web Services. *WSEAS Transaction on Computers*, 3(4):1136–1141, 2004. Obtained from `http://www.wseas.us/e-library/conferences/austria2004/papers/482-350.pdf`.

[10] Natee Artaiam and Twittie Senivongse. Enhancing Service-Side QoS Monitoring for Web Services. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 765–770, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Luciano Baresi, Sam Guinea, Marco Pistore, and Michele Trainotti. Dynamo + Astro: An Integrated Approach for BPEL Monitoring. In *Proceedings of the 2009 IEEE International Conference on Web Services*, ICWS '09, pages 230–237, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Don Box, Luis Felipe Cabrera, Craig Critchley, Francisco Curbera, Donald Ferguson, Steve Graham, David Hull, Gopal Kakivaya, Amelia Lewis, Brad Lovering, Peter Niblett, David Orchard, Shivajee Samdarshi, Jeffrey Schlimmer, Igor Sedukhin, John Shewchuk, Sanjiva Weerawarana, and David Wortendyke. Web Services Eventing (WS-Eventing). W3C Member Submission, W3C, March 2006. `http://www.w3.org/Submission/2006/SUBM-WS-Eventing-20060315/`.

[13] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. W3C Note, W3C, May 2000. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`.

[14] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, Francesco Perfetto, and Maria Villani. Service Composition (re)Binding Driven by Application-Specific QoS. In Asit Dan and Winfried Lamersdorf, editors, *Service-Oriented Computing - ICSOC 2006*, volume 4294 of *Lecture Notes in Computer Science*, pages 141–152. Springer Berlin / Heidelberg, 2006.

[15] P. Carvalho, S.R. Lima, A. Ferreira, E. Freitas, and F. Leitao. Providing Cost-effective QoS Monitoring in Multiservice Networks. In *Proceedings of the 5th Euro-NGI conference on Next Generation Internet networks*, NGI '09, pages 1 –8, july 2009.

[16] Tony Chau, Vinod Muthusamy, Hans-Arno Jacobsen, Elena Litani, Allen Chan, and Phil Coulthard. Automating SLA modeling. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, CASCON '08, pages 10:126–10:143, New York, NY, USA, 2008. ACM.

[17] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, W3C, March 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[18] James Clark and Steve DeRose. XML Path Language (XPath). W3C Recommendation, W3C, November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116/`.

[19] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI Version 3.0.2. UDDI Spec Technical Committee Draft, OASIS, October 2004. `http://uddi.org/pubs/uddi-v3.0.2-20041019.htm`.

[20] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour. Establishing and Monitoring SLAs in Complex Service Based Systems. In *Proceedings of the 2009 IEEE International Conference on Web Services*, ICWS 2009, pages 783 –790, july 2009.

[21] Doug Davis, Ashok Malhotra, Katy Warr, and Wu Chou. Web Services Metadata Exchange (WS-MetadataExchange). W3C Working Draft, W3C, August 2010. `http://www.w3.org/TR/2010/WD-ws-metadata-exchange-20100805`.

[22] Jos de Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, Birgitta König-Ries, Jacek Kopecky, Rubén Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna, and Michael Stollberg. Web Service Modeling Ontology (WSMO). W3C Member Submission, W3C, June 2005. `http://www.w3.org/Submission/2005/SUBM-WSMO-20050603/`.

[23] Jos de Bruijn, Dieter Fensel, Uwe Keller, Michael Kifer, Holger Lausen, Reto Krummenacher, Axel Polleres, and Livia Predoiu. Web Service Modeling Language (WSML). W3C Member Submission, W3C, June 2005. `http://www.w3.org/Submission/2005/SUBM-WSMO-20050603/`.

[24] DeMichiel, Linda. JSR 317: Java Persistence API, Version 2.0. Technical report, Sun Microsystems, Java Persistence 2.0 Expert Group, 2008. `http://www.jcp.org/en/jsr/detail?id=317`.

[25] Glen Dobson and Alfonso Sanchez-Macian. Towards Unified QoS/SLA Ontologies. In *Proceedings of the IEEE Services Computing Workshops*, pages 169–174, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Schahram Dustdar and Martin Treiber. A View Based Analysis on Web Service Registries. *Distributed and Parallel Databases*, 18:147–171, September 2005.

[27] ISO/IEC 14977 - Extended BNF. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`. Last visited 10/27/11.

[28] H.H. Elazhary, S.S. Gokhale, and R.A. Ammar. An efficient QoS distribution monitoring scheme. In *Proceedings of the 10th IEEE Symposium on Computers and Communications*, ISCC 2005, pages 813 – 818, june 2005.

[29] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, July 2004.

[30] Esper - API Reference. `http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/api.html`. Last visited 10/26/11.

[31] Esper - Complex Event Processing. `http://esper.codehaus.org/`. Last visited 05/11/11.

[32] Esper - EPL Reference: Clauses. `http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/epl_clauses.html`. Last visited 10/26/11.

[33] Esper - EPL Reference: Patterns. http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/event_patterns.html. Last visited 10/26/11.

[34] Esper - EPL Reference: Views. http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/epl-views.html. Last visited 10/26/11.

[35] Esper - Extension and Plug-in. http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/extension.html. Last visited 10/26/11.

[36] Esper - Performance-Related Information. http://esper.codehaus.org/esper/performance/performance.html. Last visited 05/30/11.

[37] Esper - Performance Wiki. http://docs.codehaus.org/display/ESPER/Esper+performance. Last visited 10/26/11.

[38] Esper - Processing Model. http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/processingmodel.html. Last visited 11/02/11.

[39] Esper - Reference Documentation. http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/. Last visited 10/26/11.

[40] Esper - Solution Patterns. http://esper.codehaus.org/tutorials/solution_patterns/solution_patterns.html. Last visited 11/02/11.

[41] Esper - Technology Overview. http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/technology_overview.html. Last visited 10/26/11.

[42] Li Fei, Yang Fangchun, Shuang Kai, and Su Sen. A Policy-Driven Distributed Framework for Monitoring Quality of Web Services. In *Proceedings of the 2008 IEEE International Conference on Web Services*, ICWS '08, pages 708 –715, sept. 2008.

[43] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[44] Frag - Creating DSLs with Frag. http://www.infosys.tuwien.ac.at/Staff/zdun/frag-doc/DSLToc.html. Last visited 06/09/11.

[45] Frag Documentation. http://www.infosys.tuwien.ac.at/Staff/zdun/frag-doc/. Last visited 06/09/11.

[46] GlassFish - Open Source Application Server. http://glassfish.java.net/. Last visited 05/04/11.

[47] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, December 1996.

[48] IBM. http://www.ibm.com/. Last visited 05/23/11.

[49] IBM. WSLA: Web Service Level Agreements. http://www.research.ibm.com/wsla/. Last visited 05/05/11.

[50] Java API for XML Web Services 2.0 Final Release. http://download.oracle.com/otndocs/jcp/jaxws-2_0-fr-eval-oth-JSpec/. Last visited 10/27/11.

[51] Java Management Extensions (JMX). http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html. Last visited 11/07/11.

[52] JavaServer Facelets. http://facelets.java.net/. Last visited 08/24/11.

[53] JavaServer Faces Technology. http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html. Last visited 08/22/11.

[54] Yuming Jiang, Chen-Khong Tham, and Chi-Chung Ko. Challenges and approaches in providing QoS monitoring. *International Journal of Network Management*, 10:323–334, November 2000.

[55] Li jie Jin, Li jie Jin, Vijay Machiraju, Vijay Machiraju, Akhil Sahai, and Akhil Sahai. Analysis on service level agreement of web services. Technical report, HP Laboratories, 2002.

[56] JMeter. http://jakarta.apache.org/jmeter/. Last visited 10/24/11.

[57] Radu Jurca, Boi Faltings, and Walter Binder. Reliable QoS monitoring based on client feedback. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 1003–1012, New York, NY, USA, 2007. ACM.

[58] Kavantzas, Nickolas and Burdett, David and Ritzinger, Gregory and Fletcher, Tony and Lafon, Yves and Barreto, Charlton. Web Services Choreography Description Language Version 1.0. W3C Candidate Recommendation, W3C, November 2005. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/.

[59] Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11:57–81, March 2003.

[60] Dirk Krafzik, Karl Banke, and Dirk Slama. *Enterprise SOA: service-oriented architecture best practices*. Pearson Education, Inc., 2005.

[61] Fei Li, Fangchun Yang, Kai Shuang, and Sen Su. Q-Peer: A Decentralized QoS Registry Architecture for Web Services. In *Proceedings of the 5th international conference on Service-Oriented Computing*, ICSOC '07, pages 145–156, Berlin, Heidelberg, 2007. Springer-Verlag.

[62] N.W. Lo and Chia-Hao Wang. Web services QoS evaluation and service selection framework - a proxy-oriented approach. In *Proceedings of the 2007 IEEE Region 10 Conference*, TENCON 2007, pages 1 –5, 2007.

[63] Davide Lorenzoli and George Spanoudakis. EVEREST+: run-time SLA violations prediction. In *Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing*, MW4SOC '10, pages 13–18, New York, NY, USA, 2010. ACM.

[64] David Luckham and Roy Schulte. Event Processing Glossary - Version 1.1. glossary, Event Processing Technical Society, July 2008. `http://complexevents.com/wp-content/uploads/2008/08/epts-glossary-v11.pdf`.

[65] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[66] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Web Service Level Agreement (WSLA) Language Specification. Technical report, IBM Corporation, January 2003. `http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf`.

[67] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, and Rebekah Metz. Reference Model for Service Oriented Architecture 1.0. OASIS Standard, OASIS, October 2006. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html`.

[68] E. Michael Maximilien and Munindar P. Singh. A Framework and Ontology for Dynamic Web Services Selection. *IEEE Internet Computing*, 8:84–93, September 2004.

[69] Francis McCabe, David Booth, Christopher Ferris, David Orchard, Mike Champion, Eric Newcomer, and Hugo Haas. Web Services Architecture. W3C Note, W3C, February 2004. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`.

[70] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37:316–344, December 2005.

[71] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Advanced event processing and notifications in service runtime environments. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 115–125, New York, NY, USA, 2008. ACM.

[72] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Comprehensive QoS monitoring of Web services and event-based SLA violation detection. In *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, MWSOC '09, pages 1–6, New York, NY, USA, 2009. ACM.

[73] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo. *IEEE Transactions on Services Computing*, 3:193–205, July 2010.

[74] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken SOA triangle: a software engineering perspective. In *Proceedings of the 2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, IW-SOSWE '07, pages 22–28, New York, NY, USA, 2007. ACM.

[75] Microsoft Excel - Spreadsheet - Office.com. `http://office.microsoft.com/en-us/excel/`. Last visited 09/01/11.

[76] C. Momm, M. Gebhart, and S. Abeck. A Model-Driven Approach for Monitoring Business Performance in Web Service Compositions. In *Proceedings of the fourth International Conference on Internet and Web Applications and Services*, ICIW '09, pages 343 –350, may 2009.

[77] MySQL :: The world's most popular open source database. `http://www.mysql.com/`. Last visited 08/23/11.

[78] NTP: The Network Time Protocol. `http://ntp.org/`. Last visited 08/12/11.

[79] Ernst Oberortner. *Monitoring Quality of Service in Service-oriented Systems: Architectural Design and Stakeholder Support*. PhD thesis, Vienna University of Technology, February 2010.

[80] Oracle. Java RMI. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`. Last visited 05/05/11.

[81] Michael P. Papazoglou. *Web services: principles and technology*. Prentice Hall, 2007.

[82] Mike P. Papazoglou. Service -Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, WISE '03, pages 3–, Washington, DC, USA, 2003. IEEE Computer Society.

[83] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16:389–415, July 2007.

[84] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38 –45, November 2007.

[85] Randall Perrey and Mark Lycett. Service-Oriented Architecture. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, SAINT-W '03, pages 116–, Washington, DC, USA, 2003. IEEE Computer Society.

[86] Franco Raimondi, James Skene, and Wolfgang Emmerich. Efficient online monitoring of web-service SLAs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 170–180, New York, NY, USA, 2008. ACM.

[87] Omer Rana, Martijn Warnier, Thomas B. Quillinan, and Frances Brazier. Monitoring and Reputation Mechanisms for Service Level Agreements. In *Proceedings of the 5th international workshop on Grid Economics and Business Models*, GECON '08, pages 125–139, Berlin, Heidelberg, 2008. Springer-Verlag.

[88] T. Raty, M. Karinsalo, T. Heikkila, and M. Sihvonen. Comparison of SECNM's distributed and end-to-end QoS monitoring, regarding a TLS and an unsecured connection. In *Proceedings of the 14th International Conference on Advanced Computing and Communication*, ADCOM 2006, pages 167 –172, dec. 2006.

[89] RichFaces Project Page - JBoss Community. `http://www.jboss.org/richfaces`. Last visited 08/22/11.

[90] Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Probabilistic QoS and Soft Contracts for Transaction-Based Web Services Orchestrations. *IEEE Transactions on Services Computing*, 1:187–200, October 2008.

[91] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services*, pages 205–212, Washington, DC, USA, 2006. IEEE Computer Society.

[92] SCALEA-G. `http://www.dps.uibk.ac.at/projects/scaleag/`. Last visited 05/09/11.

[93] F. Schulz. Towards Measuring the Degree of Fulfillment of Service Level Agreements. In *Proceedings of the 3rd international conference on Intercultural collaboration*, volume 3 of *ICIC 2010*, pages 273 –276, june 2010.

[94] Serenity Project. `http://www.serenity-project.org/`. Last visited 05/12/11.

[95] The SLAng SLA Language. `http://uclslang.sourceforge.net/`. Last visited 10/27/11.

[96] George Spanoudakis, Christos Kloukinas, and Khaled Mahbub. The runtime monitoring framework of serenity. In *Security and Dependability for Ambient Intelligence*, volume 45 of *Advances in Information Security*, pages 213–237. Springer US, 2009.

[97] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56:91–99, February 2001.

[98] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34:42–47, December 2005.

[99] W. Taha. Plenary talk III Domain-specific languages. In *Proceedings of the 2008 International Conference on Computer Engineering & Systems*, ICCES 2008, pages xxiii –xxviii, November 2008.

[100] The Apache Software Foundation. Apache Axis. `http://axis.apache.org/axis/`. Last visited 05/05/11.

[101] The Frag Language. `http://frag.sourceforge.net/`. Last visited 05/30/11.

[102] The Haskell Programming Language. `http://www.haskell.org/`. Last visited 05/31/11.

[103] Niko Thio and Shanika Karunasekera. Automatic Measurement of a QoS Metric for Web Service Recommendation. In *Proceedings of the 2005 Australian conference on Software Engineering*, pages 202–211, Washington, DC, USA, 2005. IEEE Computer Society.

[104] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller. A concept for QoS integration in web services. In *Proceedings of the Fourth international conference on Web information systems engineering workshops*, WISEW'03, pages 149–155, Washington, DC, USA, 2003. IEEE Computer Society.

[105] M. Tian, A. Gramm, H. Ritter, and J. Schiller. Efficient Selection and Monitoring of QoS-Aware Web Services with the WS-QoS Framework. In *Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '04, pages 152–158, Washington, DC, USA, 2004. IEEE Computer Society.

[106] Ioan Toma, Douglas Foxvog, and Michael C. Jaeger. Modeling QoS characteristics in WSMO. In *Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, MW4SOC '06, pages 42–47, New York, NY, USA, 2006. ACM.

[107] Vladimir Tosic, Babak Esfandiari, Bernard Pagurek, and Kruti Patel. On Requirements for Ontologies in Management of Web Services. In Christoph Bussler, Richard Hull, Sheila McIlraith, Maria Orlowska, Barbara Pernici, and Jian Yang, editors, *Web Services, E-Business, and the Semantic Web*, volume 2512 of *Lecture Notes in Computer Science*, pages 237–247. Springer Berlin / Heidelberg, 2002.

[108] Vladimir Tosic, Bernard Pagurek, Kruti Patel, Babak Esfandiari, and Wei Ma. Management Applications of the Web Service Offerings Language (WSOL). *Information Systems*, 30(7):564–586, 2005.

[109] Vladimir Tosic, Kruti Patel, and Bernard Pagurek. WSOL - Web Service Offerings Language. In Christoph Bussler, Richard Hull, Sheila McIlraith, Maria Orlowska, Barbara Pernici, and Jian Yang, editors, *Web Services, E-Business, and the Semantic Web*, volume 2512 of *Lecture Notes in Computer Science*, pages 57–67. Springer Berlin / Heidelberg, 2002.

[110] Hong-Linh Truong, Robert Samborski, and Thomas Fahringer. Towards a Framework for Monitoring and Analyzing QoS Metrics of Grid Services. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, E-SCIENCE '06, pages 65–, Washington, DC, USA, 2006. IEEE Computer Society.

[111] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36, June 2000.

[112] Asir S Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Ümit Yalçinalp. Web Services Policy 1.5 - Framework. W3C Recommendation, W3C, September 2007. `http://www.w3.org/TR/2007/REC-ws-policy-20070904/`.

[113] Changzhou Wang, Guijun Wang, Haiqin Wang, Alice Chen, and Rodolfo Santiago. Quality of Service (QoS) Contract Specification, Establishment, and Monitoring for Service Level Management. In *Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops*, EDOCW '06, pages 49–, Washington, DC, USA, 2006. IEEE Computer Society.

[114] Guijun Wang, Changzhou Wang, Alice Chen, Haiqin Wang, Casey Fung, Stephen Uczekaj, Yi-Liang Chen, Wayne Guthmiller Guthmiller, and Joseph Lee. Service Level Management using QoS Monitoring, Diagnostics, and Adaptation for Networked Enterprise Systems. In *Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, EDOC '05, pages 239–250, Washington, DC, USA, 2005. IEEE Computer Society.

[115] Qianxiang Wang, Yonggang Liu, Min Li, and Hong Mei. An Online Monitoring Approach for Web services. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*, volume 1 of *COMPSAC 2007*, pages 335 –342, july 2007.

[116] Ke Xu, Xiaoqi Zhang, Meina Song, and Junde Song. Research on SLA management model in service operation support system. In *Proceedings of the 5th International Conference on Wireless communications, networking and mobile computing*, WiCOM'09, pages 4912–4915, Piscataway, NJ, USA, 2009. IEEE Press.

[117] M.H. Zadeh and M.A. Seyyedi. Qos monitoring for web services by Time Series Forecasting. In *Proceedings of the 3rd IEEE International Conference on Computer Science and Information Technology*, volume 5 of *ICCSIT 2010*, pages 659 –663, july 2010.

[118] Liangzhao Zeng, Hui Lei, and Henry Chang. Monitoring the QoS for Web Services. In *Proceedings of the 5th international conference on Service-Oriented Computing*, ICSOC '07, pages 132–144, Berlin, Heidelberg, 2007. Springer-Verlag.