QuaLa – User Guide

A major requirement for many contemporary service-oriented business systems is to comply to contracts and agreements, such as Service Level Agreements (SLA). SLAs are contracts between service providers and service consumers that assure that service consumers get the service they paid for and that the service fulfills the SLA's requirements, such as availability, accessibility, or performance. Service providers need to know what they can promise within SLAs and what their IT infrastructure can deliver. To validate SLAs, mainly the services' performance-related Quality-of-Service (QoS) properties services are collected and utilized during the services runtime.

The Quality of Service Language (QuaLa) provides the facilities for describing an SLA's performance-related QoS properties, such as availability or processing time. QuaLa is separated into two sub-languages. One language, the high-level QuaLa, is tailored for experts of the QoS domain and provides constructs for specifying the services' QoS constraints that should not be violated during the system's runtime. The high-level QoS constraints need additional technical details, so that we can generate the services and the QoS monitoring infrastructure automatically. Hence, the second language, the low-level QuaLa, extends the high-level QuaLa and is tailored for technical experts for specifying the additionally needed technical details about how and where to measure the required QoS properties.

In the following we provide information about using the high- and low-level QuaLa, which technologies were used, and how to generate executable code.

1. Used Technologies

We develop QuaLa using the following technologies:

Technology/Platform	Version	Website
Java	1.5.0	http://www.java.com
Frag	0.90	http://frag.sourceforge.net
Apache CXF Framework	2.3.0	http://cxf.apache.org
Apache Ant	1.8.1	http://ant.apache.org

2. The High-Level QuaLa

The high-level QuaLa is an external DSL [Fow10] that is tailored for domain experts. Hence, the high-level QuaLa's concrete syntax is different from the language in which we implemented it, i.e., Frag[1].
In Listing 1 we illustrate the QuaLa's concrete syntax using the Extended Backus-Naur Form (EBNF).

```
sla-specification =
      sla-name '{'
            [service-name '{' qos-constraints '}']*
      '}'

qos-constraints = rule '=>' action

rule = constraint [logical-operator ['(']? Constraint [')']?]

constraint = qos-property operator predicate

qos-property = 'Availability' | 'ProcessingTime'

operator = '<' | '<=' | '>' | '>='

predicate = number unit

unit = '%' | 'd' | 'h' | 'm' | 's' | 'fps'

logical-operator = 'AND' | 'OR'

action = mail-action | sms-action

mail-action = 'mailto' '"' mail-address '"'

sms-action = 'smsto' '"' phone-number '"'
```

Listing 1 The high-level QuaLa's concrete syntax

For a better understanding of the high-level QuaLa, we give an in Listing 2.

```
MySLA {
      MyService {
            Availability>99% => mailto "me@myenterprise.org",
            ProcessingTime<2min => smsto "+1 234 56789"
      }
}
```

Listing 2 Using the high-level QuaLa in the WatchMe case study

In this example, we specify the QoS compliance concerns of a web service. The `MyService` web service should have an `Availability>99%`. If it's lower, then send an email to the system administrator. Also, the MyService web

[1] http://frag.sourceforge.net

service should have a `ProcessingTime<2min`, otherwise send an SMS to a specified cell phone number.

3. The Low-Level QuaLa

The low-level QuaLa syntax is equivalent to the syntax of the language workbench with which it was implemented. In our case the low-level QuaLa's concrete syntax is equivalent to Frag's syntax. For further information about Frag please refer to: http://frag.sourceforge.net

❖ Specifying the Technology's Architecture

In our work, we implement the WatchMe web services using the Apache CXF web service framework[2]. Hence, the technical experts have to specify the architecture of the Apache CXF web service framework. The message-flow between client and server as based on so called chains. Each chain consists of multiple phases in which interceptors are hooked for processing the message. In our case, we use interceptors for measuring the WatchMe QoS compliance concerns.
The low-level QuaLa is an embedded DSL [Fow10]. Its concrete syntax is equivalent to the language in which we implemented it, i.e., Frag. In Listing 3 we illustrate how the technical experts specify the CXF architecture and where to place the interceptors for measuring the QoS compliance concerns.

```
## CHAIN ##
cxf::InChain create ServerIn
## PHASES ##
cxf::InPhase create InPreInvoke
cxf::InPhase create InInvoke
## assign phases to chain ##
ServerIn phases {InPreInvoke InInvoke}

## PROCESSING TIME ##
ProcessingTime classes cxf::QoS
ProcessingTime chains ServerIn
ProcessingTime phases {InPreInvoke InInvoke}
```

Listing 3 Using the low-level QuaLa to specify the CXF architecture

In Listing 3 we illustrate how technical experts have to specify the CXF architecture. First, we define the chains and phases of the CXF web service framework. Then, we assign the phases to the chains. Afterwards we define in which phases of which chain the QoS properties have to be measured. In our example, we define that the processing-time is measured in the IN-Chain of the server between the Pre-Invoke and Invoke phases.
The architecture of the used web service framework has to be defined only once, because the QoS properties will be measured for each service

[2] http://cxf.apache.org

invocation in the same phases of the same chain. Changing the architecture implies to change the architecture description and the low-level QuaLa's language model.

❖ **Extending the High-Level Service Specifications with Technical Details**

Another utilization of the low-level QuaLa is the specification of technical details of the high-level service definition. In Listing 4 we illustrate how to add technical concerns to high-level services Login, Search, and Stream.

```
## MyService's technical details
MyService classes cxf::Service
MyService package "myservice"
MyService uri "http://localhost:5001/myservice"
MyService wsdl "http:// localhost:5001/myservice?wsdl"
MyService namespace "http://localhost/myservice "
MyService operations [list build \
     [cxf::Operation create myOp1 -name "myOp1"]
     [cxf::Operation create myOp2 -name "myOp2"]]
```

Listing 4 Specifying the web services' technical details using the low-level QuaLa

We specify for the `MyService` web service a package, an URI, the location of its WSDL-file, a namespace, an its operations. For example, the `MyService` web service has two operations – `myOp1` and `myOp2`.

4. An Example of Generating Code and Running the Services

Within the prototype, we provide an example of high- and low-level specifications of a service's QoS compliance concerns. The example contains one web service the customers can access to log into a system. The web service is called Login.

The high-level QoS specifications of the Login web service are located at:

```
./examples/login/high-level.sla
```

The low-level specifications are located at:

```
./examples/login/low-level.frag
```

For starting the QuaLa code generator, execute the following command:

```
/path/to/quala> ant quala
                -Dhigh-level="./examples/login/high-level.sla"
                -Dlow-level="./examples/login/low-level.frag"
```

Within a short time, the DSL's code generator generates the web service and the interceptors that measure the QoS properties. The generated code is placed in the `src-gen` folder. In the `src` folder, our code generator generates the skeletons for the web service implementations that have to be extended manually with the web services' behaviour. For example, the developer has to implemented the behaviour of the Login web service in:
`src/examples/login/Login.java`

Then, one can start the services by starting the service host, using Apache Ant[3]. Just execute the following command:

```
/path/to/quala> ant run
```

[3] http://ant.apache.org