FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Performance Test Language for Web Services

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Hermann Czedik-Eysenberg
Matrikelnummer 0526426

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.-Prof. Dr. Uwe Zdun
Mitwirkung: Projektass. Dipl.-Ing. Ernst Oberortner

Wien, 15. März 2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

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

Erklärung zur Verfassung der Arbeit

Hermann Czedik-Eysenberg
Ketzergasse 471/1
1230 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.

Abstract

During the development of Web services and their deployment to a specific target system, organizations need to evaluate if the Web service implementation and the hardware can meet the Quality of Service attributes specified in a Service Level Agreement. One important aspect are performance-related attributes, such as response time, throughput or scalability of the Web service. Typically tests which simulate client behavior are used to evaluate these attributes.

In this thesis the domain-specific language QoSTIL (Quality of Service Test Instrumentation Language) is introduced. It allows for the definition of tests and the composition of these tests to create complex test plans which can be used to simulate Web service client behavior and evaluate performance-related quality of service attributes of Web services. Language instances are automatically transformed to executable Java code and result presentation views. These can be used to run the defined tests and visually display the test results.

The implementation of the domain-specific language follows the Model-Driven Development approach. Both the language model (abstract syntax), which is based on the meta-model defined by the Frag modeling framework, the concrete syntax, as well as the transformation to executable code are explained in detail in the thesis. Also a review of technologies and development tools is included. These technologies and tools are used for the implementation of the domain-specific language and generation and execution of the runtime system.

To evaluate the utility and usability of the language a number of tests which are based on realistic business use cases for assessing certain performance-related quality of service attributes are defined and implemented. After the automatic transformation to executable code, the tests are run for specific Web services and the results are presented and discussed. Furthermore the thesis includes a comparison to the related work in the problem domain.

Kurzfassung

Während der Entwicklung von Web Services und deren Bereitstellung auf einem bestimmten Zielsystem sollte laufend überprüft werden, ob Web Service-Implementierung und Hardware die Qualitätsattribute, die in einem Service Level Agreement spezifiziert wurden, erfüllen können. Ein wichtiger Aspekt sind Performance-bezogene Attribute, wie Antwortzeiten, Durchsatz und die Skalierbarkeit eines Web Service. Typischerweise werden Tests verwendet, die ein bestimmtes Clientverhalten simulieren, um diese Attribute zu evaluieren.

In dieser Diplomarbeit wird die domänen-spezifische Sprache QoSTIL (Quality of Service Test Instrumentation Language) vorgestellt. Sie ermöglicht die Definition von Tests und deren Komposition zur Erstellung von komplexen Testplänen. Diese Tests können verwendet werden um Web Service-Clientverhalten zu simulieren und Performance-bezogene Qualitätsattribute zu überprüfen. Sprachinstanzen werden automatisch zu ausführbarem Java-Code und Testreport-Ansichten transformiert. Diese können dafür eingesetzt werden um die definierten Tests auszuführen und die Testresultate graphisch anzuzeigen.

Die Implementierung dieser domänen-spezifischen Sprache folgt dem Ansatz der modellgetriebenen Softwareentwicklung. Sowohl das Sprachmodell (die abstrakte Syntax), das auf dem vom Frag Modeling Framework definierten Metamodell basiert, als auch eine konkrete Syntax, sowie die Transformation zu ausführbarem Code werden im Detail erläutert. Außerdem beinhaltet die Arbeit einen Überblick der Technologien und Entwicklungswerkzeuge, die zur Implementierung der domänen-spezifischen Sprache und zur Generierung und Ausführung des Laufzeit-Systems eingesetzt wurden.

Um den Nutzen und die Bedienbarkeit der entwickelten Sprache zu evaluieren, wurde eine Anzahl von Tests, die auf realistischen Business-Anwendungsfällen basieren, für die Messung von bestimmten Performance-bezogenen Qualitätsattributen definiert und implementiert. Nach der automatischen Transformation zu ausführbarem Code, wurden die Tests auf ein bestimmtes Web Service angewendet und die Ergebnisse werden in der Arbeit präsentiert und diskutiert. Weiters enthält die Diplomarbeit einen Vergleich mit ähnlichen Arbeiten innerhalb des Forschungsgebiets.

Contents

CHAPTER 1

Introduction

In recent years the use of service-oriented architecture – and its implementation using Web services – has become one of the most popular approaches for developing distributed systems. It has been successfully applied to intra- and inter-organizationally integrate business functionality and applications. Organizations can establish agile and flexible collaborations by using well-defined services that are independent of the used computing platforms and software frameworks.

Besides the functional specifications of Web services, non-functional requirements are a major issue for all parties involved in a collaboration. Such requirements are specified in contracts and agreements, such as Service Level Agreements (see section 2.2). They are mainly expressed in terms of Quality of Service (QoS) guarantees that service providers and consumers have to fulfill. One important aspect are performance-related QoS attributes, such as response time, throughput or scalability of a Web service.

To validate such performance-related QoS attributes, they have to be evaluated using quantifiable measurements. The most common approach to obtain performance results of a given Web service is to simulate a specific client behavior and workload and observe the performance of the system. This is usually denoted as performance testing [16].

In this thesis a new domain-specific language (DSL) for defining and running Web service performance tests is introduced. A DSL is a small language tailored to be particularly expressive in a certain problem domain, here Web service performance testing. This DSL was developed following the Model-Driven Development paradigm (see section 2.4) and can be used to describe and compose performance tests using a textual syntax, which is tied to elements described in the language model.

1.1 Motivation

Today Web services are widely adopted in the industry and their providers have to work hard to stay competitive. Often, there are similar services from different providers available that can be easily interchanged by service consumers. QoS is a significant factor in defining the success of Web services and their providers. Since it directly influences the utility and usability of the service for its consumers, it plays an important role in determining the popularity of a Web service.

Both, service providers and service consumers have a strong interest in performance testing of Web services. Web service providers have to make sure, that their services can meet the QoS guarantees that are arranged in agreements and contracts. Otherwise they will face serious financial penalties and lose their credibility. Service consumers want to evaluate, if they are getting the service quality they paid for, or, if they should switch service providers. They might also want to find some arguments for possible legal actions against a contractual partner, if they suspect a breach of agreement.

The performance testing methodology should be clearly specified, reusable and easy to communicate. Therefore we want to provide a modern DSL that supports the definition and execution of performance tests. It enables developers to define such tests at an adequate level of abstraction. The implementation of the tests is automatically generated by a transformation to executable Java code. The language itself is independent of the system that is tested and its implementation technologies.

The language is mainly targeted at Web service developers that want to evaluate performance as early as possible. Usually, performance testing is conducted at the end of the development, after system integration testing. It can however also take place during the initial development of a system, which is known as Early Performance Testing [2]. The performance test cases are defined and developed together with the implementation of the system or even before the real coding starts. This helps to ensure that QoS attributes are always kept in mind. Thus, performance bottlenecks can be found early and the cost involved in solving them is reduced to a great extent.

The tests developed in the language presented here should be quick to define and easy to maintain and thereby support an early and transparent testing process. All involved developers should be able to efficiently repeat tests whenever needed. The performance results should be available as clearly structured and effortlessly accessible test reports.

Of course, the language presented here is not only usable for early testing, but can also be applied after development has finished as well as by service consumers, that want to test an existing Web service. An important area of application is also regression testing after changes have been made to an existing Web service, i.e. to try to uncover performance problems that are results of the changes.

1.2 Problem Definition

Performance testing is a way to evaluate performance-related quality attributes of a system. For Web services this can serve different purposes: A service provider might want to demonstrate that the system meets specific performance goals. Service consumers may want to compare different system to find out which one performs better. For Web service developers it is a way to better understand the performance of a system. They may want to use performance test results to determine what causes for performance degradation of their system exist and where performance bottlenecks are.

Performance testing should be differentiated from performance monitoring. Monitoring normally means to observe and measure the performance of a deployed system while it is used in normal extent by its real users. Performance testing, on the other hand, involves the generation of artificial load to produce a pre-defined workload condition. The usage of the system (number and types of requests and clients) is exactly defined in a test plan and simulated by test clients. The exact quality attributes and the way that they are measured should also be accurately documented. The test implementation should follow these exact rules and the test results should be reproducible. Normally the tests are not performed on production systems, to make sure that the performance for the real users is not affected by the test clients, and vice versa. In any case, the statistical significance of the test results should be considered.

Menascé [16] defines a performance testing methodology with seven main steps:

1. **Defining the Testing Objectives.** The purpose is the definition of the goals of the performance tests. A testing goal may be something very concrete, such as finding out the maximum number of concurrent users a Web service supports within the limits of the Service Level Agree-

ments. Or something more general, such as identifying bottlenecks in the Web service infrastructure.

2. **Understanding the Environment.** The tester has to learn everything about infrastructure, software and relevant quality attributes of the system under test.

3. **Specifying the Test Plan.** The test plan is a detailed description of the whole testing process. It should include the information which Web services and functions are tested, how they are requested and how performance is measured. Also, it should contain a definition of the quality attributes that the test evaluates and the Service Level Agreements that will be verified.

4. **Specifying the Test Workload.** In this step, the user behavior that should be simulated, has to be devised. Scripts that represent the user behavior have to be created.

5. **Setting Up the Test Environment.** This is the process of installing measurement and testing tools.

6. **Running the Tests.** The execution of the tests should follow the testing plans. The test results should be documented, including a detailed description of test parameters to allow reproduction of the test.

7. **Analyzing the Results.** Based on the data collected, the analysts should be able to determine bottlenecks that cause performance problems. An important issue when analyzing the results is to make sure that the reported measurements are coherent, i.e. that there are no errors in the measurement process.

The DSL and its runtime environment presented in this thesis were developed to support the steps beginning with the specification of the test plan (step 3) and ending with running the tests (step 6). Automatically generated reports additionally assist the last step, the analysis of the test results. The main focus of the language model is on the specification of the test workload (step 4).

Performance tests are often characterized by the intensity of the generated load [16]:

Load testing The goal is to understand the performance of the system under a specific expected load.

Stress testing In stress testing, the load is raised beyond normal usage patterns, in order to make sure that a Web service works (or at least gracefully fails) under worst-case conditions.

Spike testing By spiking the workload (short periods of time where the load is several times larger than average) it should be determined how the system is able to handle dramatic changes in load.

The performance testing language aims to include constructs to configure all of these types of workload in detail.

Another important goal of the introduced language is to make the definition of measurements and evaluations of quality attributes as flexible as possible. A service level agreement might for example contain the condition, that the service provider needs to guarantee a given average throughput, if the system is not over-utilized by more than 10 percent and that maximum response time should never exceed a certain value. There is an infinite number of such quality attributes that might be important for a specific business or application. Often different organizations require different definitions even for crucial attributes such as availability, throughput or response time. The language has to be especially flexible, when it comes to expressing them.

1.3 Organization of the thesis

The following chapter 2 contains a state of the art overview of relevant topics, such as Web services, Quality of Service and Model-Driven Development. It also shortly introduces technologies that have been applied for the development of the DSL presented in the thesis.

Chapter 3 describes in detail the design of the new language and its implementation.

The practical application of the language is discussed in chapter 4. It contains numerous examples, a usability and utility evaluation of the language and also discusses possible future language extensions.

Related work on Web service performance testing is examined in chapter 5.

The last chapter 6 is a summary and concludes the thesis.

CHAPTER 2

State of the Art

2.1 Service-Oriented Architectures and Web services

A service-oriented architecture (SOA) [3] is a way of organizing software applications and infrastructures into a set of interacting services. It provides an architectural model for the service-oriented computing paradigm [24] which utilizes services as fundamental elements for mastering the complexity of distributed applications.

A basic model is given by the classic SOA triangle shown in figure 2.1 [24]. It considers three main roles: The *service registry* is a directory where service descriptions can be published and searched. The *service provider* implements a service and publishes the service specification in a service registry. The *service requestor* (client) queries the registry to find a certain service. If found, it binds to the service endpoint and can finally use the service by invoking its operations.

There is a common set of service-level design principles for a SOA [14]:

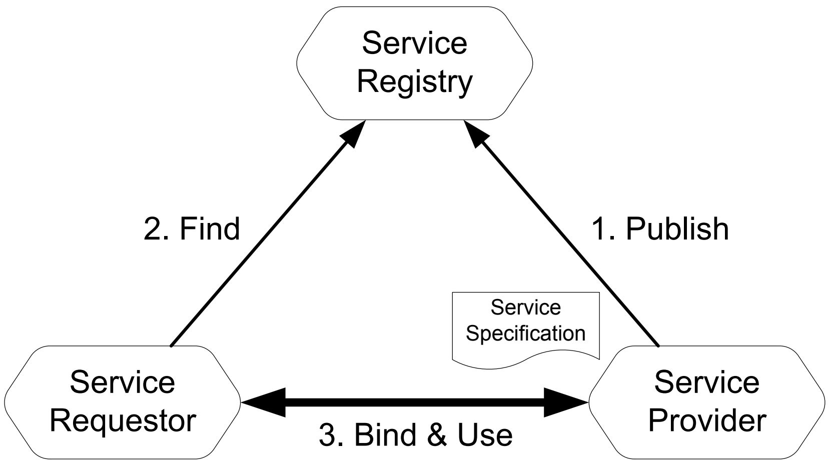


Figure 2.1: Basic SOA model.

- Services are **reusable** and **context independent**. They are designed to support potential reuse. They do not depend on the context in which they are used.
- Services share a **formal contract** (service specification). It defines the terms of information exchange and any supplemental service description information.
- Services are **loosely coupled**. In order for them to interact, they do not share anything but the service specification. This implies that clients are not restricted to one predetermined supplier.
- Services are **self-contained** and **abstract underlying logic**. The only part that is visible to the outside world is what is exposed via the service specification. The implementation details do not matter to the service client.
- Services are **composable**. They may use and compose other services. This allows logic to be represented at different levels of granularity and promotes the creation of abstraction layers. Services that assemble ex-

isting services from possibly multiple service providers are referred to as composite services [24].

- Services should be **stateless**. They should not be required to manage state information, since that can interfere with their ability to remain loosely coupled.
- Services are **discoverable**. They should publish their descriptions and allow them to be discovered by potential clients.
- Services have a **network-addressable interface**. They must support remote requests.

Of the principles described above, autonomy, loose coupling, abstraction and the need for a formal contract can be considered the core principles that form the baseline foundation for SOAs [20].

Since Services may be offered by different organizations and communicate over the Internet, they provide a distributed computing infrastructure for both intra- and cross-organization application integration and collaboration. Consequently it is essential that services are technology-neutral and the invocation mechanisms (protocols, descriptions and discovery mechanisms) comply with widely accepted open standards.

An implementation technology and umbrella term for a lot of well-established standards for SOAs are Web services [23]. Two of the most important standards associated with Web services are, the Web Services Description Language (WSDL) [37], which is used to describe Web services and their service specification, and SOAP [38], a protocol used to exchange messages between Web services and their clients.

SOAP defines the use of XML as an encoding scheme for request and response parameters typically using HTTP as a means for transport. A SOAP message consists of a mandatory body, which contains the message payload or business information, and an optional header, which specifies additional handling options and can be used by extension protocols.

2.2 Quality of Service and Service Level Agreements

Quality of Service (QoS) [20] can be defined as a set of non-functional properties that determine the quality a service offers to its clients. The many aspects of QoS important to SOAs and Web services can be organized into QoS categories, often called quality attributes. Each attribute needs to have a set of quantifiable parameters or measurements. Some of the most important quality attributes for SOA are [20, 26]:

Interoperability A measure of whether the service complies with standards.

Availability The percentage of time a service is operating.

Reliability The ability of a service to perform its required functions under stated conditions for a specified period of time.

Security Includes the existence and type of authentication mechanisms the service offers, confidentiality and data integrity of messages exchanged, non-repudiation of requests or messages, and resilience to denial-of-service attacks.

Performance A measure of the speed in completing service requests.

Scalability The ability of a SOA to function well (without degradation of other quality attributes) when the system is changed in size or in volume in order to meet users' needs. One of the major issues in scalability is how the performance of a system is affected by an increasing number of service users.

Robustness It is the degree to which a service can function correctly in the presence of invalid, incomplete or conflicting inputs.

Accuracy The error rate produced by a service.

Integrity To which degree a service can guarantee the consistency of the data it operates on.

Maintainability The ease with which a system can be run and modified if necessary. This is closely related to Deployability (the ease with which the system can be installed and run), Extensibility (the ease with which the capabilities can be extended without affecting other parts of the system), and Adaptability (the ease with which as system may be changed to fit changed requirements).

Cost A measure of cost involved in requesting the service.

Configuration Management related Some attributes are related to configuration management: Completeness (measure of the difference between the specified set of features and the implemented set of features), Stability (measure of the frequency of change related to the service in terms of its specification and/or implementation), Testability (the degree to which a system facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met), and Regulatory (how well the service is aligned with regulations).

A Service Level Agreement (SLA) [12] is a contract between service providers and service consumers that specifies a set of QoS guarantees and the obligations of the parties. Simplified example conditions that an SLA may contain include:

- The Web service should be available at least 99.9 percent of time.

- The average response time for a particular service request should not exceed 1 second.
- All traffic has to be encrypted using a highly secure industry standard encryption algorithm.

SLAs for Web services are either plain natural language documents, or instances of SLA templates that include several automatically processable fields in an otherwise natural language document, or they can even be fully expressed using flexible formal languages, such as in the WSLA framework [13].

2.3 Measuring performance-related QoS for Web services

Since this thesis focuses on performance-related QoS attributes of Web services, a set of quantifiable parameters and measurements for these attributes is defined in this section.

Peformance

The following distinguishable and relevant periods in time, a SOAP request goes through before completing a full round trip, have been defined by Rosenberg et al. [27] based on the work by Wickramage and Weerawarana [39]:

Processing time The processing time t_p defines the time needed by the service provider implementation to actually carry out the operation for a specific request. It does not include any communication overhead and is therefore the parameter with the smallest granularity.

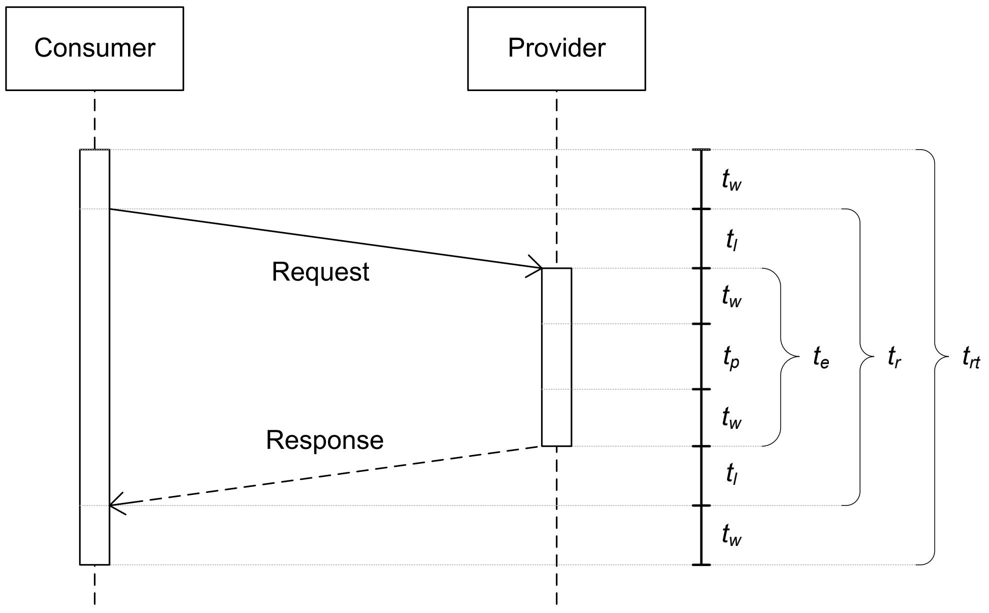


Figure 2.2: A Web service operation invocation and involved time frames.

Wrapping time The wrapping time t_w is a measure for the time that is needed to unwrap the XML structure of a received request or wrap a request before sending it to the destination.

Execution time The execution time t_e represents the whole time the service provider needs to finish processing a request. It starts with unwrapping the XML structure, then processing the result and finally wrapping the answer into a SOAP message that can be sent back to the requestor. It is simply the sum of two wrapping times and the processing time:

$$t_e = t_w + t_p + t_w.$$

Latency time The time a message needs to reach its destination over the network is called latency time t_l. It is influenced by the type of the network connection, routing, network utilization and the message size.

Response time The response time t_r is the time needed for sending a message from a given client to a service provider until the response returns back to the client. It is calculated by simply adding the network latency for each direction to the execution time:

$$t_r = t_l + t_e + t_l.$$

Round trip time The round trip time t_{rt} gives the overall time that is consumed for invoking a Web service operation. It comprises all values on both, requestor and provider side. It is calculated by adding to the response time above also the wrapping time the client needs before sending a message and after receiving the answer:

$$t_{rt} = t_w + t_r + t_w = t_w + t_l + t_w + t_p + t_w + t_l + t_w.$$

A sequence diagram showing a full Web service operation invocation with a graphical representation of all involved time frames is depicted in figure 2.2.

Oberortner et al. [19] have described patterns for measuring performance-related QoS properties in distributed systems. In this thesis their pattern of automatically generated *QoS interceptors* is used for measuring time frames involved in Web service requests at both the server- and client-side (see chapter 4).

Throughput

Throughput tp is the number of service requests a service can successfully complete over a time period. It can be calculated by the formula

$$tp = \frac{\#R}{t},$$

where $\#R$ is the number of successfully completed service requests and t is the measured time period, e.g. in seconds.

Scalability

One important performance-related aspect of scalability is that a system that is scalable has the ability to not get overloaded by a massive number of parallel requests [27]. The scalability in this respect can be evaluated by observing the time spans of single requests and the throughput of the system at increasing levels of concurrently running service requests. While the request time spans, such as processing time and round trip time, for single requests may increase, the throughput should not break down, if a service is confronted with unexpected high load. Additionally it is important how the request time spans rise, if the number of concurrent requests increases. A service where, e.g., the average round trip time increases linearly with the number of concurrent requests is considered to have better scalability than a service where the average round trip time increases quadratically or even exponentially.

2.4 Domain-Specific Languages and Model-Driven Development

Domain-specific languages (DSLs) [7] are small languages that are tailored to be particularly expressive in a certain problem domain. DSLs enable domain experts to work at higher levels of abstraction. The goal is to make modeling complex problems in a domain easier and more convenient. Furthermore they can remove the necessity to manually write schematic and recurring code by supporting automatic code generation. It has been shown that carefully executed and narrowly defined DSLs can reap an order of magnitude improvement in productivity and quality [35].

A popular approach to implementing DSLs is Model-Driven Development (MDD) [31]: The graphical or textual syntax of the DSL is tied to domain-specific modeling elements through a precisely defined language model. It has been shown that MDD-based DSLs for modeling business systems enable technical and non-technical experts to work at higher levels of abstraction [17].

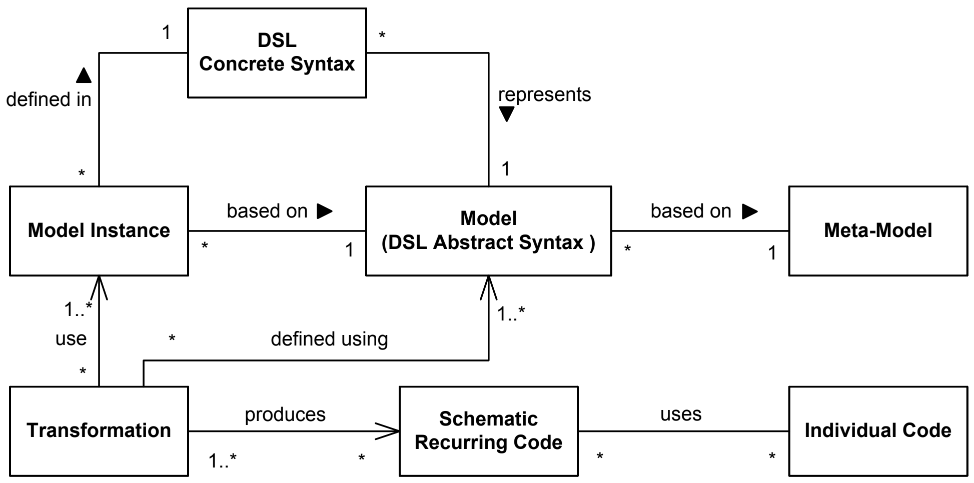


Figure 2.3: Architecture of DSLs based on MDD [18].

The general infrastructure of MDD-based DSLs is depicted in figure 2.3. The core of the DSL is the *language model*, which is also called the *DSL abstract syntax*. It defines the elements of the domain and their relationships without considering their notations. It is based on a *meta-model* which defines how the domain elements and their relations can be described. DSL users should be able to describe particular problems of their domain by defining *model instances* (based on the language model) using a familiar notation. Therefore, the *DSL concrete syntax* defines either a textual or graphical language for defining model instances in a form that is suitable for the stakeholders using the DSL.

A *transformation* maps model concepts to code or other output and can therefore be defined when model instances should be transformed into instances of some other model and, ultimately, to an executable language. Typically DSLs are used to automatically generate *schematic and recurring code* of an application that is developed for a particular problem domain. A tool that uses a transformation to generate executable code from a model instance is called a generator. If necessary, the automatically generated code can be extended with manually written *individual code* (or maybe also code that was generated by using some other DSL).

Tolvanen [35] divides the process of implementing DSLs into four phases:

1. Identifying abstractions and how they work together. It is important to describe things in problem domain terms instead of implementation concepts.

2. Working out the language model. Generally major domain concepts should be mapped to modeling language objects, while other concepts are captured as object properties, connections, sub-models, or links to models in other languages.

3. Creating the visual representation for the language. Defining a domain-specific notation makes models much easier to create, read an maintain.

4. Defining the generators. They transform model instances into code for interpretation or compilation into an executable.

The implementation of the DSL described in this thesis can also be divided in these four phases, as presented in the following chapter 3.

2.5 Applied Technologies

This section briefly describes the tools, languages and software frameworks that were used for the implementation of the system presented in this thesis.

Frag Modeling Framework

Frag [41, 40] is a dynamic programming language, specifically designed for enabling MDD and building DSLs. It supports both embedded DSLs (also called internal DSLs) and external DSLs [6]. An embedded DSL is an extension to an existing programming language and uses the syntactic elements of the underlying language, hosting the DSL. An external DSL is defined in a different format than the intended host language and can use all kinds of syntactical elements.

The Frag Modeling Framework (FMF) is a package that is included with Frag and can be used to build a language model for a DSL. It is similar to the modeling frameworks found in other model-driven language workbenches (e.g. Eclipse Modeling Framework, Microsoft DSL Tools) and the representation of the language model can easily be mapped to existing modeling languages, e.g. a UML class diagram. The meta-model defined by the FMF allows to specify language models in Frag using well-known concepts, such as classes, which can have typed and untyped attributes, relationships between classes (associations, compositions, aggregations, and inheritance), as well as extensions using stereotypes and enumerations.

Apache CXF

Apache CXF [33] can be used to build and develop services using frontend programming APIs, such as JAX-WS [9]. One of its primary focuses is the support of Web services standards.

The CXF framework will be used in this thesis for implementing an example Web service infrastructure that will be subject to performance-related quality of service tests.

Apache Ant

Apache Ant [32] is a Java library and command-line tool for building applications. It uses an XML file (typically named `build.xml`) for the description of the build process of a project in terms of targets and tasks.

It is used for building the whole implementation of the system presented in this thesis.

JavaCC

Java Compiler Compiler (JavaCC) [1] is, according to its authors, the most popular parser generator for Java-based applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar.

In this thesis a JavaCC-based parser is used for the implementation of the external DSL syntax.

JBoss Seam

JBoss Seam [11] is a powerful open source development platform for building rich Internet applications in Java. It is used in this thesis for building a Web user interface for displaying test result reports.

CHAPTER 3

Language Design and Implementation

This chapter describes the design and implementation of the QoSTIL (Quality of Service Test Instrumentation Language). The purpose of this new DSL is to enable its users to define and compose tests to create complex test plans which can be used to simulate Web service client behavior and evaluate performance-related quality of service attributes of Web services. The core concept of the language are *tests* which can be transformed to executable code, run against a particular Web service or service infrastructure. After the execution the tests return test reports, which include the test results, i.e. measurements of numeric QoS attributes and possibly also evaluations of Boolean assertions about such attributes, in a human readable form. Complex test behavior can be described by composed tests which execute other tests using loops or in parallel. The results of such test execution sequences can easily be merged using *aggregators* which implement statistical measures such as average, variance, and median.

3.1 Language Model and Semantics

Figure 3.1 depicts the core language model which contains the main language element ***Test*** and related classes.

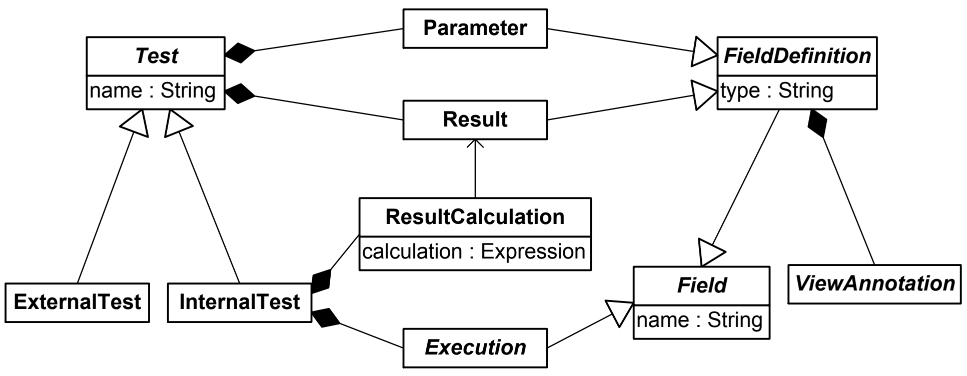


Figure 3.1: QoSTIL model: Core.

Each ***Test*** has a string attribute *name* which is used to identify it. Tests can have parameters (**Parameter**) and results (**Result**). Both parameters and results are subtypes of ***FieldDefinition*** and indirect subtypes of ***Field*** and therefore have a *type* (e.g. integer number) and a *name*. All fields can be seen as variable slots that are used to store and access values by name. Parameters are used for input values that a test needs when it is executed and have to be filled by a test caller. Results have to be filled by the test implementation during the test execution. The test report that is returned after the execution of a test contains all parameters and results.

Test is an abstract class which has two concrete subclasses: **ExternalTest** and **InternalTest**. An external test does not have its implementation specified using this language, it is only a declaration of a test interface. The test behavior and calculations have to be implemented in the hosting language (Java). Internal tests on the other hand are completely specified using elements from the language model, i.e. executions (***Execution***) and result calculations (**ResultCalculation**).

Executions specify which other tests should be called and run during the execution of some test. As subtypes of ***Field*** they also have a *name* which can later on be used to access their values. The value of an execution is its result, i.e. a test report (or a sequence of test reports) of the test that was executed. The subtypes of the abstract class ***Execution*** are explained in detail later (see figure 3.2).

Result calculations associate each test result with an expression which specifies how the result value is calculated after all executions have completed. In an internal test each **Result** is associated with exactly one **ResultCalculation**. Expressions are further explained later using figure 3.3.

Field definitions (parameters and results) can be associated with an optional set of view annotations (***ViewAnnotation***). They specify how these fields will be displayed in the test report that is generated after a test was run. For the set of available concrete view annotations see figure 3.5 later in this section.

The translation from a language model given by a class diagram to a FMF language model using Frag syntax is straightforward. Code listing 3.1 shows how some concepts from the core language model (figure 3.1) can be expressed in the FMF.

By using the Frag language more complex constraints can be expressed, such as that the number of result calculations always needs to be the same as the number of results for an internal test. These constraints are not contained in the example above. The following parts of the language model are depicted using class diagrams only.

All executions (see figure 3.2) are associated to a **TestRun**. It specifies which other test should be executed (using the attribute *test*) and it has a set of

```
FMF::Class create Test -attributes {
  name String
}
FMF::Class create InternalTest -superclasses Test
FMF::Class create ExternalTest -superclasses Test

FMF::Class create Parameter -superclasses FieldDefinition
FMF::Class create Result -superclasses FieldDefinition

FMF::Composition create Test-Parameter -ends {
  {Test -roleName test -multiplicity 0
    -navigable true -aggregatingEnd true}
  {Parameter -roleName parameters -multiplicity * -navigable true}
}
FMF::Composition create Test-Result -ends {
  {Test -roleName test -multiplicity 0
    -navigable true -aggregatingEnd true}
  {Result -roleName results -multiplicity * -navigable true}
}

FMF::Class create ResultCalculation -attributes {
  result Result
  calculation Expression
}
FMF::Composition create InternalTest-ResultCalculation -ends {
  {InternalTest -roleName test -multiplicity 0
    -navigable true -aggregatingEnd true}
  {ResultCalculation -roleName resultCalculations
    -multiplicity * -navigable true}
}

FMF::Class create Execution -superclasses Field -attributes {
  run TestRun
}
FMF::Composition create InternalTest-Execution -ends {
  {InternalTest -roleName test -multiplicity 0
    -navigable true -aggregatingEnd true}
  {Execution -roleName executions -multiplicity *
    -navigable true}
}
```

Code Listing 3.1: QoSTIL core language model described in the syntax of the FMF (some concepts).

parameter setters, which specify using expressions to what values the parameters of the other test are set before it is called. Exactly one **ParameterSetter** is needed for each parameter of the other test.

There are two types of executions: A **SingleExecution** will execute the other test only once. Repeated executions (***RepeatedExecution***) will execute

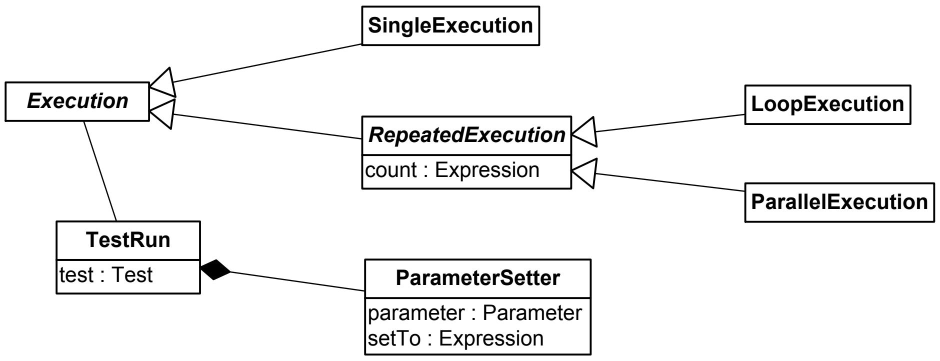


Figure 3.2: QoSTIL model: Executions.

the test a fixed number of times (given by the attribute *count*). Again there are two variants: **LoopExecution** specifies to execute the other test in a serial fashion, one run after the other. The next iteration only begins after the first one is completed. **ParallelExecution** on the other hand specifies that the same test should be executed for the given number of times in parallel, all runs starting at the same time.

The value of an execution is the result of the test that was executed. For single executions, this is a test report, which contains all parameter and result values of the test. For repeated executions it is a sequence (or array) of test reports.

Expressions (see figure 3.3) can be used to calculate some value. The subtypes of ***Value*** are language elements that have an obvious value. They are fields (parameters, executions and results) and literals (e.g. the number 0). Furthermore there a field selectors (**FieldSelector**) which can be used to only select the value of a particular field of a test report given by an expression (typically this expression will contain an execution, because its value is a test report).

Operators can be used to build arbitrary complex expressions. There are

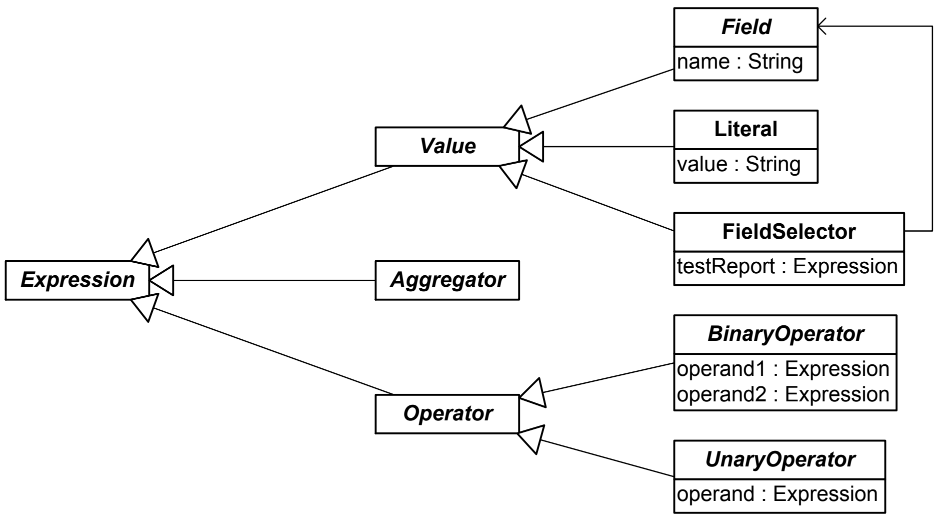


Figure 3.3: QoSTIL model: Expressions.

both binary and unary operators. Their concrete subtypes (which are not listed in the figure) are all Boolean (and, or, not ...), numeric (addition, multiplication ...) and relational (equals, greater than ...) operators which are supported by the hosting language Java.

Aggregators are functions which take a sequence (or array) of test reports (typically the result of a repeated execution) as an argument and associate it with some new value. The available aggregators are given in figure 3.4.

The argument of an aggregator is given by the expression-typed attribute *testReportArray*. Field aggregators (***FieldAggregator***) have an additional attribute *aggregationField* which associates the aggregator with a particular test report field which is selected in all test reports that it aggregates over. The concrete aggregators and their functions are:

ArrayConcatAggregator Selects the given aggregation field (which needs to be array typed) for each test report in the given test report array and concatenates all values building a new large array.

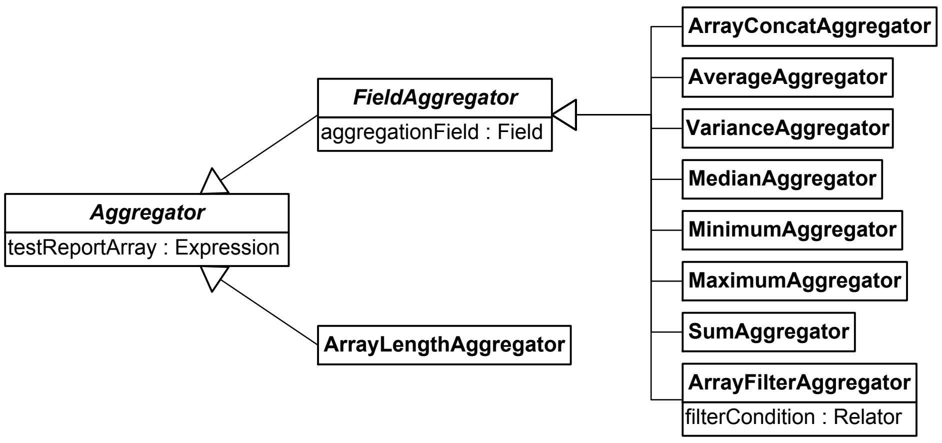


Figure 3.4: QoSTIL model: Aggregators.

AverageAggregator Calculates the average value over all values of the given aggregation field (which needs to be numerically typed) for all test reports in the given test report array.

VarianceAggregator Calculates the variance (mean squared deviation).

MedianAggregator Calculates the median value.

MinimumAggregator Calculates the minimum value.

MaximumAggregator Calculates the maximum value.

SumAggregator Calculates the sum of all selected (numerically typed) fields.

ArrayFilterAggregator Builds a new test report array by selecting only those reports where the value of the given aggregation field matches some condition given by the *filterCondition* attribute. The available relators are all relational operators supported by the hosting language Java (equals, greater than ...).

ArrayLengthAggregator Calculates the length of the given test report array.

There are of course more possible aggregator functions. The aggregators described here were identified as the most important ones for the language use cases discussed in the following chapter 4.

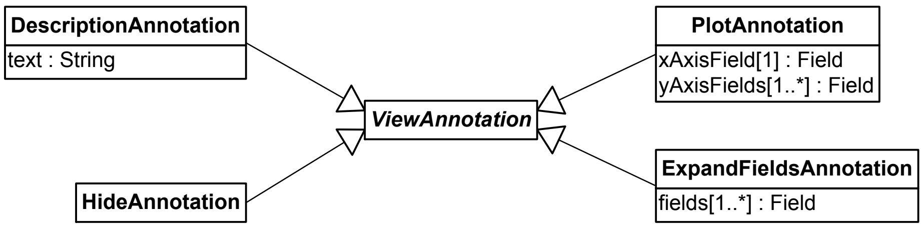


Figure 3.5: QoSTIL model: View annotations.

The available concrete view annotation subtypes that can be used to configure how test fields (parameters and results) are displayed in the test report, are depicted in figure 3.5. Using **DescriptionAnnotation** a descriptive text can be added to parameters and results. It will be displayed in the test report next to the name of the field and can be used to make a test report easier understandable. If a field is annotated using an instance of **HideAnnotation** it will not be displayed in the test report. This can be used for test implementation related fields that are not relevant for the interpretation of the test results by users.

PlotAnnotation can be used for presenting the data obtained from repeated executions in the test report using a plot in the form of curves on a Cartesian plane. The annotated field needs to be test report array typed, in other words its value needs to be an array of test reports. The attribute *xAxisField* defines which field from each of the test reports in the test report array will be used for selecting the set of X values. The Y values are given by *yAxisFields*. It is possible to give more than one Y-axis field, if multiple curves should be displayed in the same plot.

ExpandFieldsAnnotation can be used to display some of the fields of some other test report (the value of the field that this annotation is used on) directly in the test report of the current test. Normally only a link to the other test report is displayed, but using this annotation the fields selected using the attribute *fields* (and their values) are presented on the same page. This annotation is only allowed for fields that are test report or test report array typed. In the latter case the selected fields of each test report from the array will be displayed in a table.

3.2 Language Syntax

By defining the language model in the FMF we have the possibility to also create language instances using the Frag syntax. The language is then used as an embedded (or internal) DSL. Such language instances are directly linked to the language model which brings a lot of benefits, e.g. constraints such as association multiplicities or attribute types can be checked automatically. But language instances specified using the Frag syntax are quite long and not easily readable. For example, this is how a simple QoSTIL-expression containing a subtraction of two literals could be expressed in the FMF syntax:

```
[SubtractionOperation create
  -operand1 [Literal create -value "10"]
  -operand2 [Literal create -value "5"]]
```

Therefore an external, shorter syntax was defined and implemented as a notation for the language based on the language model. The syntax is designed to look familiar to programmers knowing Java (or other C-like languages). Together with this syntax, QoSTIL can be used as an external DSL, because it is not any more directly embedded in a host language.

Code listing 3.2 is a grammar in an extended Backus-Naur Form that describes the QoSTIL syntax.

```
File          = ( InternalTest | ExternalTest )*

InternalTest  = 'test' ID '{' IntlTestBody '}'
ExternalTest  = 'external' 'test' ID '{' ExtlTestBody '}'

IntlTestBody  = ( ParameterDecl ';' )*
                ( ExecutionDef  ';' )*
                ( ResultDef     ';' )*
ExtlTestBody  = ( ParameterDecl ';' )*
                ( ResultDecl    ';' )*

ParameterDecl = [Annots] 'parameter' Typename ID
ResultDecl    = [Annots] 'result'    Typename ID
ResultDef     = [Annots] 'result'    Typename ID '=' Expression

ExecutionDef  = [Repeated] 'execution' ID ':' TestRun
Repeated      = ( 'loop' | 'parallel' ) '(' Expression ')'
TestRun       = ID '(' [ ParamSetter ( ',' ParamSetter )* ] ')'
ParamSetter   = ID '=' Expression

Annots        = ( DescAnnot | HideAnnot | ExpandAnnot | PlotAnnot )*
DescAnnot     = '@Description' '(' STRING ')'
HideAnnot     = '@Hide'
ExpandAnnot   = '@ExpandFields' '(' ID ( ',' ID )* ')'
PlotAnnot     = '@Plot' '(' ID '|' ID ( ',' ID )* ')'
```

Code Listing 3.2: Grammar of the QoSTIL syntax.

In this grammar all capitalized words are non-terminals, characters between single quotes (') are terminals, `( ... | ... )` indicates a choice, the asterisk `( ... )*` specifies a repetition for zero or more times and `[ ... ]` is an option. `ID` is a special non-terminal for identifiers which is used for all alphanumeric names. `STRING` stands for all string literals between double quotes (").

The top-level non-terminal `File` is used for compilation units and can include multiple internal and external test definitions. All tests start with the keyword `test` followed by their name and the test body inside curly brackets. External tests are distinguished from internal tests using the modifier `external`. Test bodys can include parameter declarations (`ParameterDecl`) and result definitions for internal tests (`ResultDef`) or result declarations for external tests (`ResultDecl`). Internal tests also allow execution definitions (`ExecutionDef`).

Parameter declarations begin with the keyword `parameter` (after optional view annotations), are typed using a `Typename` and have a name. The non-terminal `Typename` is not explicitly given in the grammar above. It includes all primitive Java types (bool, int, double ...), strings (String) and the (array) test report types of tests defined in this language. Test report names are always composed of the name of a test and the character sequence "`Report`" (e.g. "`MyTestReport`"). Test report array types can be defined by appending square brackets (e.g. "`MyTestReport[]`").

Result declarations are equivalent to parameter declarations. Result definitions additionally include an expression after an equality sign, which specifies how the result value is calculated.

Due to space constraints the expression syntax is left out in the above grammar. It is generally very similar to the Java expression syntax: Variables (fields) such as parameter, result and execution names are simply accessed using their name. The syntax for numeric and string literals, as well as for all operations is equivalent to the Java syntax. Field selectors for test reports can be expressed using a dot-notation (e.g. `testReport.field1`). The notation of aggregators is very similar to the method call syntax of Java. E.g. the length of a test report array typed variable `testReportArray` can be calculated by the expression `testReportArray.length()`. An example for a field aggregator looks like this: `testReportArray.average(field1)`. Filter aggregators additionally include the relation expression, e.g. `testReportArray.filter(field1 > 10)`.

Execution definitions are introduced by the keyword `execution` and also have a name. They can be modified by the keywords `loop` and `parallel` if the execution is supposed to be repeated multiple times. The number of repetitions is given by an expression in brackets following the modifier. After a colon, execution definitions have to contain a `TestRun` which specifies the name of the other test that should be executed followed by a pair of brackets which can optionally contain a list of parameter setters (`ParamSetter`). Parameter setters begin with the name of the other test's parameter that should be set, followed by an expression specifying the value, after an equality sign.

Optionally view annotations (`Annots`) can be added to parameter and result declarations/definitions. The view annotation syntax is heavily influenced by the Java annotation syntax.

An example of a full test definition to get a better idea of the syntax (without view annotations) is contained as code listing 3.3.

```
test MyTest {

  parameter double myParam1;
  parameter int myParam2;

  loop(10) execution otherTestExecutions:
      OtherTest(otherTestParam1 = myParam1 * 100);

  result int myResult1 = 5 * (myParam2 + 10);
  result OtherTestReport[] myResult2 = otherTestExecutions;
  result double myResult3 =
      otherTestExecutions.average(otherTestResult1);
}
```

Code Listing 3.3: Example QoSTIL test definition.

The example defines a test named `MyTest` which declares two typed parameters (`myParam1` and `myParam2`) and an execution named `otherTestExecutions` of `OtherTest` that will be executed in a loop for 10 times with a value for the parameter `otherTestParam1` given by an expression. Furthermore there are three results (`myResult1`, `myResult2` and `myResult3`) including expressions that specify how their values are calculated. The value of `myResult2` is simply the result of the repeated execution, i.e. an array of test reports, and therefore needs to have the type `OtherTestReport[]`. The result of `myResult3` is the average value of the field named `otherTestResult1` from each of the test reports in the array. This assumes of course that the test definition of `OtherTest` does include this field.

Code listing 3.4 is the same example which has been extended with some view annotations.

```
test MyTest {

  @Hide
  parameter double myParam1;

  @Description("The second parameter of the test")
  parameter int myParam2;

  loop(10) execution otherTestExecutions:
      OtherTest(otherTestParam1 = myParam1 * 100);

  @Hide
  result int myResult1 = 5 * (myParam2 + 10);

  @Plot(otherTestResult1 | otherTestResult2)
  @ExpandFields(otherTestResult1, otherTestResult2)
  result OtherTestReport[] myResult2 = otherTestExecutions;

  result double myResult3 =
      otherTestExecutions.average(otherTestResult1);
}
```

Code Listing 3.4: Example QoSTIL test definition (with view annotations).

The view annotations are used to control how the visual representation of the test report that is generated when this test is executed should look like. Here it is stated that `myParam1` and `myResult1` should be hidden, i.e. not be included in the test report. A description for humans is added to `myParam2`. For the test reports in `myResult2` a plot will be generated. The values from the `otherTestResult1` fields are plotted on the X-axis and the values from `otherTestResult2` on the Y-axis. Furthermore the same fields and their values will also be included in the test report using a textual representation, as stated by the `@ExpandFields` annotation.

The implementation of a parser for the syntax defined above has been developed using JavaCC (see section 2.5). The parser generates a parse tree which is then mapped to the elements from the language model and results in a language instance. The mapping component is implemented in Frag using the FMF and therefore generates language instances that are directly linked to the language model (which was also specified using the FMF). This has the advantage that language constraints are checked automatically and all features of the FMF can easily be used, such as code generation (see the following sections).

3.3 Code Generation

To make tests executable, the language instances are automatically transformed to Java code. This code generation component was also implemented using the FMF: It reads all tests from a language instance and produces executable Java classes that often contain schematic and recurring code that is tedious and error-prone to write manually.

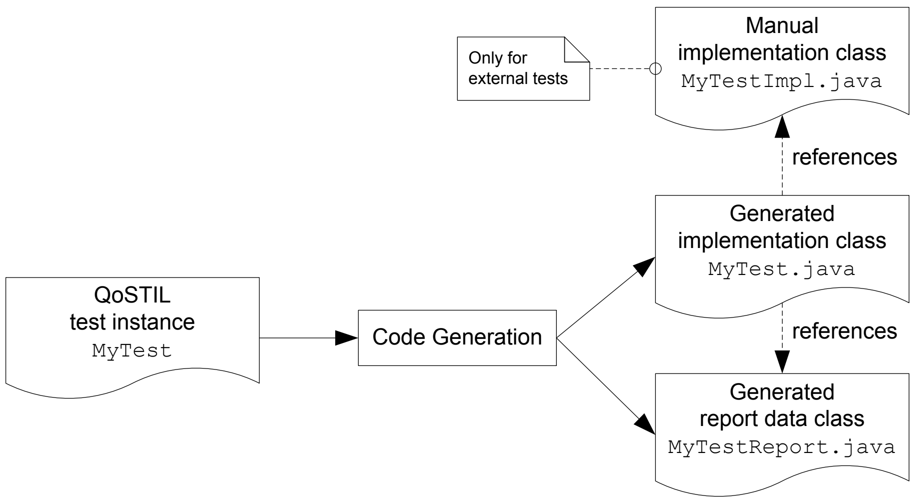

Only for external tests
Manual implementation class MyTestImpl.java
references
Generated implementation class MyTest.java
QoSTIL test instance MyTest
Code Generation
references
Generated report data class MyTestReport.java

Figure 3.6: QoSTIL code generation and involved documents.

Each test is automatically transformed to two Java classes (see figure 3.6). The first class (**implementation class**) which implements the behavior of the test, simply uses the test name specified in the model instance as its name (e.g. `MyTest`). It always has a method named `run` which can be used to execute the test. For external tests an additional class with the same interface is expected and has to be implemented manually. It has to be named alike plus "Impl" (e.g. `MyTestImpl`). It is referenced in the generated run method and it leads to a compilation error if it does not exist.

The second class (**report data class**) is used to store both the parameter and result values of the test. It is called just like the first class with the string "Report" appended (e.g. `MyTestReport`). Parameter and result values are saved using instance fields and can be accessed using setter and getter methods. An instance of this class (already containing all test parameter values) has to be passed as an argument to the run method of the implementation class and will be returned by the same method, then also containing the test result values. The report data class is additionally extended using JAXB annotations which make it possible to serialize test reports to XML data. This is used to save test reports to present them to the language users after the test executions.

3.4 Report View Generation

In this section we present the report view generation component of the implemented system. The goal of the this component is to generate a report template for each test that will be used to visually present the test output to the language user after the test execution.

Normally the test report should include all parameter and result field values of the executed test. Using view annotations the test report display can be configured, by adding descriptions, hiding certain fields, defining plots and expanding some fields of sub test reports.

In the implemented system, after a test was executed, the test report data (containing all parameter and result values of this particular test run) is serialized to XML and stored in a database associated with a unique identifier

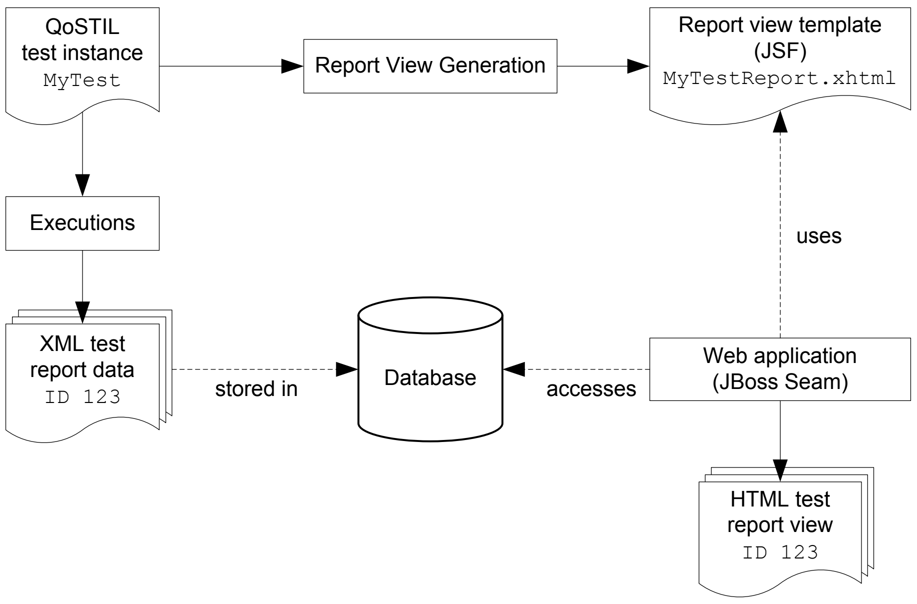

QoSTIL test instance MyTest
Report View Generation
Report view template (JSF) MyTestReport.xhtml
Executions
uses
XML test report data ID 123
stored in
Database
accesses
Web application (JBoss Seam)
HTML test report view ID 123

Figure 3.7: QoSTIL report view generation and involved documents.

(see figure 3.7). Later on the data can be accessed by the language users through a Web interface which presents it as an HTML page according to the configuration by view annotations. Therefore, besides the Java classes that are needed to execute a test and store the test data, a report view template for each test is automatically generated (also depicted in figure 3.7), that is used to present the reports of particular test executions to the language users. The generated view templates are based on the JavaServer Faces (JSF) [10] technology, which is supported by popular Java web application frameworks, such as JBoss Seam which was used here.

Plots are automatically rendered client-side by the Javascript plotting library *flot* [21].

CHAPTER 4

Using the Language

To show the utility and usability of the QoSTIL a set of prototypical performance tests have been implemented as example language instances. The definition of these tests is based on existing methods for assessing relevant performance-related quality of service attributes and uses the metrics as described in chapter 2, section 2.3. There are tests for measuring both low-level attributes such as (average) response times for single and repeated Web service requests and more complex ones such as throughput and scalability of a Web service.

The implementation of the most basic test case described below that simply performs a single Web service request, interprets the response and uses both client- and server-side interceptors to measure the involved time frames, is not generated from the DSL presented here, but is based on the **low-level QuaLa** defined by Oberortner et al. [18, 36]. All the other more complex tests described here and their implementations are completely defined using the QoSTIL, without the need to use other programming languages.

To prove the practical applicability of QoSTIL tests and show the high value of performance tests, the results of the most complex test given here (a scalability test) will be discussed and interpreted in detail in section 4.2.

At the end of this chapter (section 4.3) there is some discussion of possible future extensions of the QoSTIL that could make the language more powerful and simpler to use.

4.1 Language Instances

There are four main performance tests for Web services listed in this section (and also some intermediate helper tests). They all have been defined using the QoSTIL in a very generic and parameterizable way. The simplest test case is the *Basic Request Test*. Each one of the more complex tests is always based on the previous simpler test, leading to a test hierarchy. The *Generic Parallel Load Test* (indirectly) reuses the *Basic Request Test*, the *Throughput Test* executes the *Generic Parallel Load Test*, and finally the most advanced test defined here, the *Scalability Test*, is based on the *Throughput Test*.

Basic Request Test

The purpose of the *Basic Request Test* is to perform a single Web service request and return whether the request was successful and how much time was needed to execute it. In particular the time here should be measured as both the complete roundtrip time and the more fine-grained processing time (see section 2.3).

```
external test WSRequestTest {
  result double processingTime;
  result double roundtripTime;
  result boolean successful;
}
```

Code Listing 4.1: QoSTIL test instance: `WSRequestTest`.

Such a simple test can be defined in the QoSTIL as in code listing 4.1. This test definition only specifies the three test results. It is an external test, which means that the actual implementation of the test behavior is not specified using the QoSTIL.

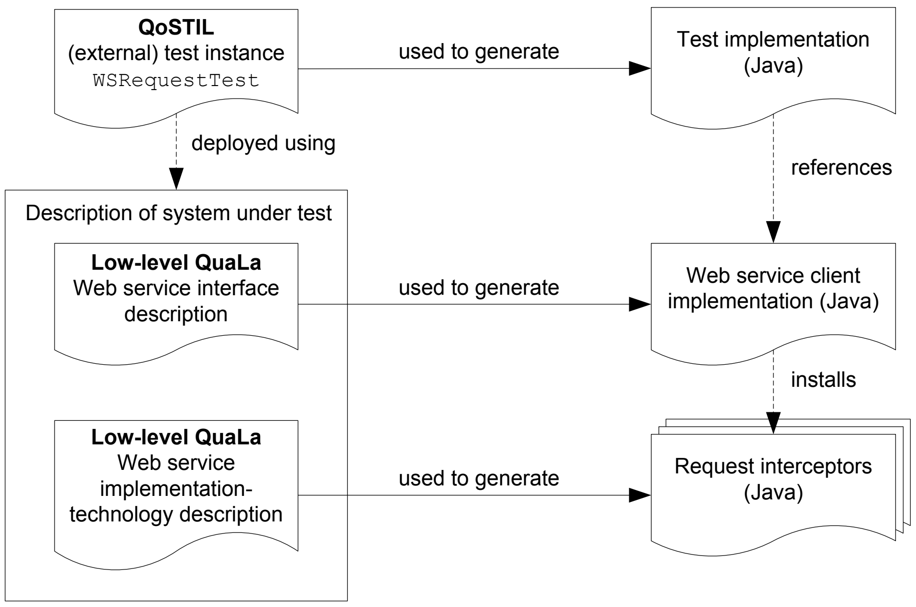

QoSTIL
(external) test instance
WSRequestTest
used to generate
Test implementation (Java)
deployed using
references
Description of system under test
Low-level QuaLa
Web service interface description
used to generate
Web service client implementation (Java)
installs
Low-level QuaLa
Web service implementation-technology description
used to generate
Request interceptors (Java)

Figure 4.1: Interaction of the QoSTIL with the low-level QuaLa.

As explained above, the implementation is supposed to be added using another DSL, the low-level QuaLa [18, 36]. The connection of the QoSTIL presented in this work and the previously defined low-level QuaLa is depicted in figure 4.1. The low-level QuaLa is used for specifying the system under test, i.e. the Web service that should be tested.

The first part of this specification is a description of the *Web service interface* including how the Web service can be accessed, what operations it supports and what parameters these operations have.

The second part is a description of the *Web service implementation technology*. It has to include all information that is needed to model the implementation-specific concerns, e.g. how to measure the roundtrip time in a particular Web service engine. In the developed prototype a description of the open-source Apache CXF Web service framework (see section 2.5) is included. Using the model provided by the low-level QuaLa the requirements can be modeled as follows [18]: The communication between service client and service provider is based on message-flows. Each message-flow consists of a number of phases, where each phase can contain handlers (or interceptors) for measuring QoS values. For instance, the handler for measuring the roundtrip time is associated to two certain phases of the message flow on the client side.

The low-level QuaLa specification of the system under test can be used to automatically create an executable Java *Web service client implementation* and the *request interceptors* for the described Web service technology, which are needed to measure the (low-level) QoS attributes of interest. The Web service client implementation makes sure that the necessary request interceptors are installed on both the client- and server-side.

The Java *test implementation* that is generated from the QoSTIL `WSRequestTest` defined above, references the Web service client implementation and uses it for performing a single Web service request. Through a defined interface it accesses the measured QoS values (here processing time, roundtrip time and successful) and stores them in the specified result fields.

The `WSRequestTest` defined above measures the performance of a Web service request only in terms of processing and roundtrip time, as opposed to all the time frames described in section 2.3. But a measurement of the involved network and protocol overhead (the sum of all latency and wrapping times) can easily be obtained by subtracting the processing time from the roundtrip time.

This can be implemented in the QoSTIL using a simple wrapper for `WSRequestTest`, here called `SingleRequestTest` (code listing 4.2). This internal test simply executes the `WSRequestTest` once, copies all results

```
test SingleRequestTest {

  execution request: WSRequestTest();

  result double processingTime = request.processingTime;
  result double roundtripTime = request.roundtripTime;
  result boolean successful = request.successful;
  result double latencyAndWrappingTime =
    roundtripTime - processingTime;

}
```

Code Listing 4.2: QoSTIL test instance: `SingleRequestTest`.

(`processingTime`, `roundtripTime`, `successful`) unchanged and calculates a new result `latencyAndWrappingTime` by subtracting the processing time from the roundtrip time.

Generic Parallel Load Test

The *Basic Request Test* described above can be used to measure basic QoS attributes for single Web service requests. But to determine significant performance statements about Web services, realistic client behavior has to be simulated, especially multiple clients that access a Web service in parallel.

The goal is to develop a *Generic Parallel Load Test* that can be used to simulate a fixed number of parallel clients, where each of the clients repeatedly performs Web service requests. The results and basic measurements of each executed request should be returned for further processing (e.g. statistical analysis).

To develop this performance test, first an intermediate test was implemented using the QoSTIL. The `RepeatedRequestTest` definition (code listing 4.3) simply runs the `SingleRequestTest` described above in a loop.

The purpose of this test is to run a Web service request repeatedly for a fixed number of times. This number is specified by the parameter `repeats`.

```
test RepeatedRequestTest {

  parameter int repeats;

  loop(repeats) execution requests: SingleRequestTest();

  result SingleRequestTestReport[] singleRequestTestReports =
    requests;

}
```

Code Listing 4.3: QoSTIL test instance: `RepeatedRequestTest`.

The test definition uses a loop execution to repeatedly run the previously defined `SingleRequestTest` in a serial fashion. The reports of these executions are returned as the test result `singleRequestTestReports`. Consequently this result is `SingleRequestTestReport[]`-typed, which restricts its value to be an array of `SingleRequestTest` reports.

```
test ParallelRequestTest {

  parameter int parallelUsers;
  parameter int requestsPerUser;

  parallel(parallelUsers) execution requests:
    RepeatedRequestTest(repeats = requestsPerUser);

  result SingleRequestTestReport[] singleRequestTestReports =
    requests.concat(singleRequestTestReports).
      filter(successful == true);

}
```

Code Listing 4.4: QoSTIL test instance: `ParallelRequestTest`.

The `RepeatedRequestTest` is used to simulate a single client that repeatedly accesses a Web service. Using another test this test is parallelized to simulate multiple equivalent clients (see code listing 4.4). This `ParallelRequestTest` has a parameter `parallelUsers` for specifying how many parallel clients should be simulated. This parameter gives the number of times the parallel execution should run the `RepeatedRequestTest`. The value of the parameter `requestsPerUser` is used as an input value for the `repeats` parameter of each execution.

The only result (singleRequestTestReports) of this test is again just an array of SingleRequestTest reports. The value of this result is calculated by simply concatenating all result arrays of the Repeated-RequestTest executions (using an **ArrayConcatAggregator**). Additionally in this example the results are filtered to only contain the reports of successful requests (using an **ArrayFilterAggregator**).

Throughput Test

The ParallelRequestTest explained above is a general test that can be used to simulate parallel users that repeatedly invoke a Web service. The results of this test can be used for evaluating all kinds of performance-related QoS attributes of Web services.

One example for a test that directly uses the ParallelRequestTest for calculating important QoS attributes is the ThroughputTest, which was developed as the next example for the usage of the QoSTIL and is contained as code listing 4.5.

This test has two defined parameters (parallelUsers and requests-PerUser) which are directly used for input values for the Parallel-RequestTest, that is executed by this test. The result of this execution is stored in a field named requests. The result calculation definitions showcase how QoSTIL aggregators can be used to calculate statistical properties of datasets. The average, variance, minimum and maximum of the processing times of all Web service request that were performed during the execution of the ParallelRequestTest are calculated and stored in test results. Equivalently these statistical measures are also calculated for the roundtrip times and the latency and wrapping times (but left out in the definition above due to space constraints).

The test also returns the total number of requests that were performed

```
test ThroughputTest {

  parameter int parallelUsers;
  parameter int requestsPerUser;

  execution requests:
    ParallelRequestTest(parallelUsers = parallelUsers,
                        requestsPerUser = requestsPerUser);

  result double processingTimeAvg =
    requests.singleRequestTestReports.average(processingTime);
  result double processingTimeVar =
    requests.singleRequestTestReports.variance(processingTime);
  result double processingTimeMedian =
    requests.singleRequestTestReports.median(processingTime);
  result double processingTimeMin =
    requests.singleRequestTestReports.min(processingTime);
  result double processingTimeMax =
    requests.singleRequestTestReports.max(processingTime);

  // equivalent average, variance, median, minium and maximum
  // calculations for roundtripTime and latencyAndWrappingTime

  result int totalRequestsCount =
    parallelUsers * requestsPerUser;
  result int successfulRequestsCount =
    requests.singleRequestTestReports.length();
  result double requestsPerMinute =
    (successfulRequestsCount * 60000.0) / testTime;
  result double requestsPerMinutePerUser =
    requestsPerMinute / parallelUsers;

  result double testTime = testTime;
}
```

Code Listing 4.5: QoSTIL test instance: `ThroughputTest`.

(`totalRequestsCount`) and how many of them were successful (`suc-cessfulRequestsCount`).

The value of the result `requestsPerMinute` is the average number of request per minute that could be successfully completed by the Web service with the given number of parallel users. This is of course a measurement for the throughput of the Web service and the reason for the name of this test. By dividing this value by the number of parallelUsers (result `requestsPerMinutePerUser`) an estimate for the throughput from the viewpoint of a single client is also obtained.

Please note the usage of the special field `testTime` that is available in the generated runtime code of every QoSTIL test and contains the time the test needed for running (only execution time, without the calculation of result values).

Scalability Test

The *Scalability Test* is the most advanced test presented here as an example for the usage of the QoSTIL.

It can be used to determine the scalability of a Web service by observing its throughput at increasing levels of concurrently running users. This can be achieved by repeatedly running the *Throughput Test* from the previous section, with increasing values for the parameter `parallelUsers`.

A flexible *Scalability Test* can is defined in code listing 4.6. Unlike the previous tests, this test definition also demonstrates the usage of QoSTIL *view annotations* for parameters and results.

The `@Description` annotations add a descriptive text to all parameters and results. Therefore they do not need to be further explained here.

Using a *loop execution* the *Throughput Test* is repeatedly executed (serially and not in parallel of course) for a fixed number of times (given by the parameter `repeats`). The values for the parameters of the *Throughput Test* are automatically calculated for each iteration. Please note the usage of the special variable `iteration` that can be used in all definitions of repeated executions. It can be used to access the current iteration count, beginning with 1.

The reports of the executed *Throughput Tests* are stored unmodified in the result `throughputTestReports` for further processing or interpretation by the language user. By using `@Plot` annotations a visual presentation of

```
test ScalabilityTest {

  @Description("Number of executed throughput tests")
  parameter int repeats;

  @Description("Number of parallel users in the first iteration")
  parameter int startParallelUsers;

  @Description("Increase of parallel users per iteration")
  parameter int parallelUsersIncrease;

  @Description("Number of requests per executed throughput test")
  parameter int requestsPerTest;

  loop(repeats) execution reports:
    ThroughputTest(parallelUsers = startParallelUsers * iteration,
      requestsPerUser =
        requestsPerTest / (startParallelUsers * iteration));

  @Plot(parallelUsers |
    processingTimeAvg, processingTimeMedian,
    processingTimeMin, processingTimeMax,
    roundtripTimeAvg, roundtripTimeMedian,
    roundtripTimeMin, roundtripTimeMax,
    latencyAndWrappingTimeAvg, latencyAndWrappingTimeMedian,
    latencyAndWrappingTimeMin, latencyAndWrappingTimeMax)
  @Plot(parallelUsers | requestsPerMinute, requestsPerMinutePerUser)
  @ExpandFields(parallelUsers, totalRequestsCount,
    successfulRequestsCount, requestsPerMinute,
    requestsPerMinutePerUser)
  @Description("Reports of the executed throughput tests")
  result ThroughputTestReport[] throughputTestReports = reports;

  @Description("Total execution time")
  result double testTime = testTime;

}
```

Code Listing 4.6: QoSTIL test instance: `ScalabilityTest`.

the most important test results will be generated in the test report. The first plot shows the number of parallel users on the X-axis and the average (median / minimum / maximum) processing (roundtrip / latency and wrapping) time on the Y-axis. The second plot has the same X-axis but visualizes the throughput (request per minute / request per minute per user) on the Y-axis.

Additionally the most important results of each *Throughput Test* are also directly displayed textually in the report of a *Scalability Test* because of the

`@ExpandFields` annotation.

An example for a report that is generated by running the *Scalability Test* is included in the following section.

4.2 Evaluation

To illustrate the usage and functionality of the tests described in the previous section, a test demonstration setup was developed. All tests were executed for a particular example Web service. In this section the test environment and some test results are presented and discussed.

Test Environment

Figure 4.2 depicts the setup of the simple demonstration test environment. The *test client* is a standard laptop with a 2.20-GHz dual-core CPU and 4 gigabytes of RAM. It executes the test runtime environment that was automatically generated from the QoSTIL tests described previously. It also hosts the database that is used for storing the test reports. The *system under test* is a standard PC too, equipped with a 2.0-GHz dual-core CPU and 1 gigabyte of RAM. It hosts the example Web service that should be tested. The computers are connected via a 1gbit/s Ethernet switch.

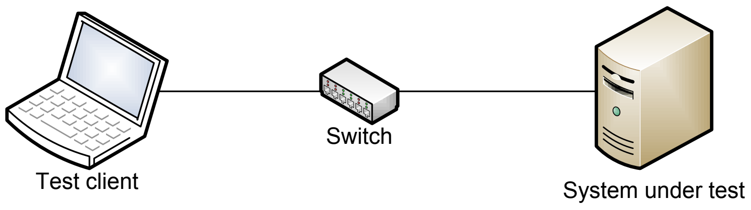


Figure 4.2: Test environment setup.

The example Web service used for the demonstration described here, is called *PiApproximator* and supports a single operation that returns the number pi approximated to a fixed number of decimal places. To simulate some processing time and CPU load, the implementation of the operation uses an approximation function with 600,000 iterations and never caches the result.

Test Execution Results

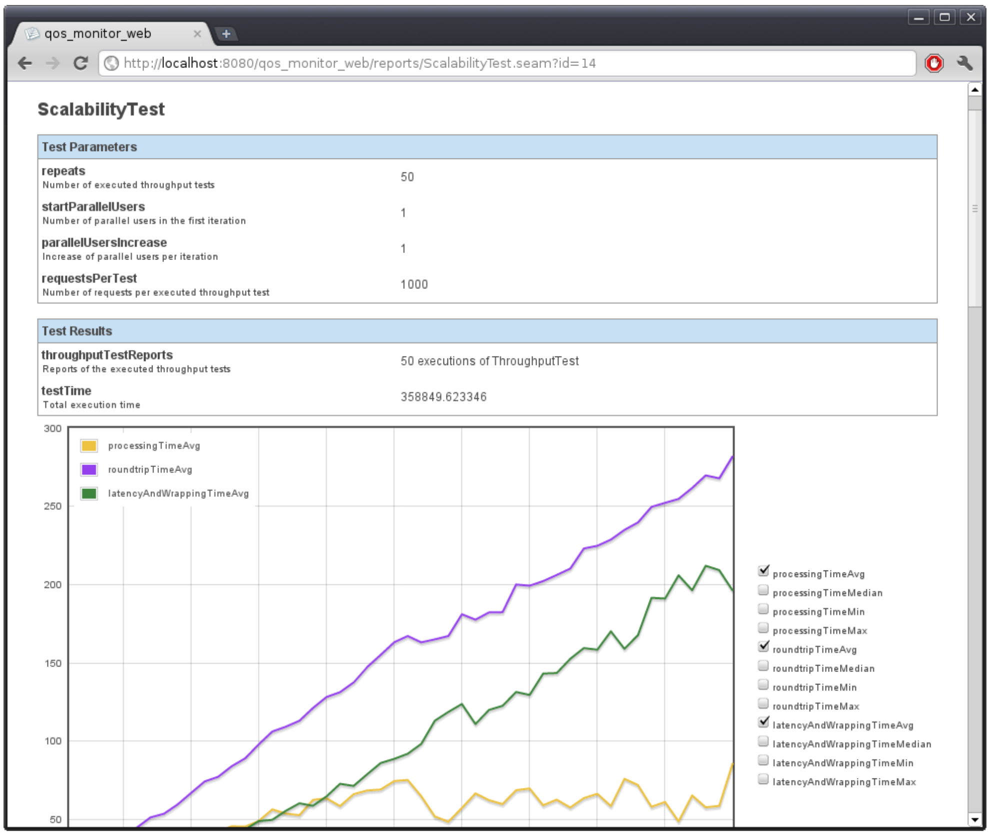
Using the given test environment all previously defined tests were executed with multiple sets of values for the input parameters. Since the *Scalability Test* is the most complex test, a particular execution of this test should be discussed in detail here.

repeats	50
startParallelUsers	1
parallelUsersIncrease	1
requestsPerTest	5000

Table 4.1: Parameter values for the executed *Scalability Test*.

Table 4.1 contains the parameter values that were used for this particular execution. The test should be repeated for 50 times, beginning with one parallel user and increasing the number of parallel users by one in each iteration. For every test iteration 5000 Web service request should be performed. Hence, in the last iteration there are 50 parallel users and each user invokes the Web service a hundred times.

Running the *Scalability Test* using the given parameter values generates a test report. The test report is automatically stored in a database and can later on be retrieved using the Web user interface. Figure 4.3 is a screenshot of a part of the generated HTML test report. It can be seen that the report lists the


qos_monitor_web
http://localhost:8080/qos_monitor_web/reports/ScalabilityTest.seam?id=14

ScalabilityTest

Test Parameters	
repeats Number of executed throughput tests	50
startParallelUsers Number of parallel users in the first iteration	1
parallelUsersIncrease Increase of parallel users per iteration	1
requestsPerTest Number of requests per executed throughput test	1000

Test Results	
throughputTestReports Reports of the executed throughput tests	50 executions of ThroughputTest
testTime Total execution time	358849.623346

Figure 4.3: Report of the executed *Scalability Test* in the Web user interface (screenshot).

test parameter and result values. Furthermore plots for particular result value sets are generated, as specified by the annotations in the definition of the test.

For further discussion of the test results, the generated plots have been extracted from the report and are contained in this work as figures 4.4, 4.5 and 4.6.

Figure 4.4 depicts the average roundtrip, average processing and average latency & wrapping time for the *PiApproximator* Web service while the number of users accessing the service simultaneously is steadily increased.

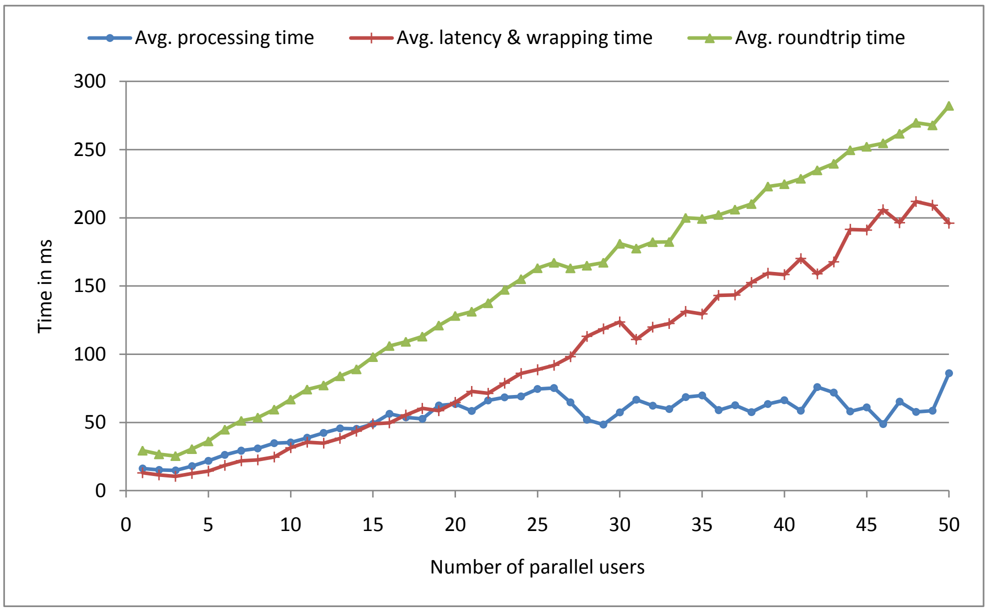


Figure 4.4: *Scalability Test* report: Average processing, latency & wrapping and roundtrip time at increasing levels of parallel users.

Up to the maximum number of 50 parallel users, all averaged request times seem to increase very linearly. E.g., the average latency & wrapping time starts at about 15 milliseconds if there are no parallel requests (only one parallel user), rises up to about 90 milliseconds for 25 parallel users and reaches about 200 milliseconds for the maximum number of 50 users. This corresponds to an average latency & wrapping time increment of about 3.7 milliseconds for each added virtual user.

The processing time on the other hand exhibits a slower growth (only about 1.7 milliseconds for each added virtual user). A possible conclusion is that the performance of the Web service is more heavily influenced by the network and protocol overhead and not so much by the actual implementation of the pi approximation function. This is already a very interesting result of the *Scalability Test* for this Web service system. Using this new knowledge the QoSTIL user can draw consequences what needs to be optimized to make the Web service more scalable.

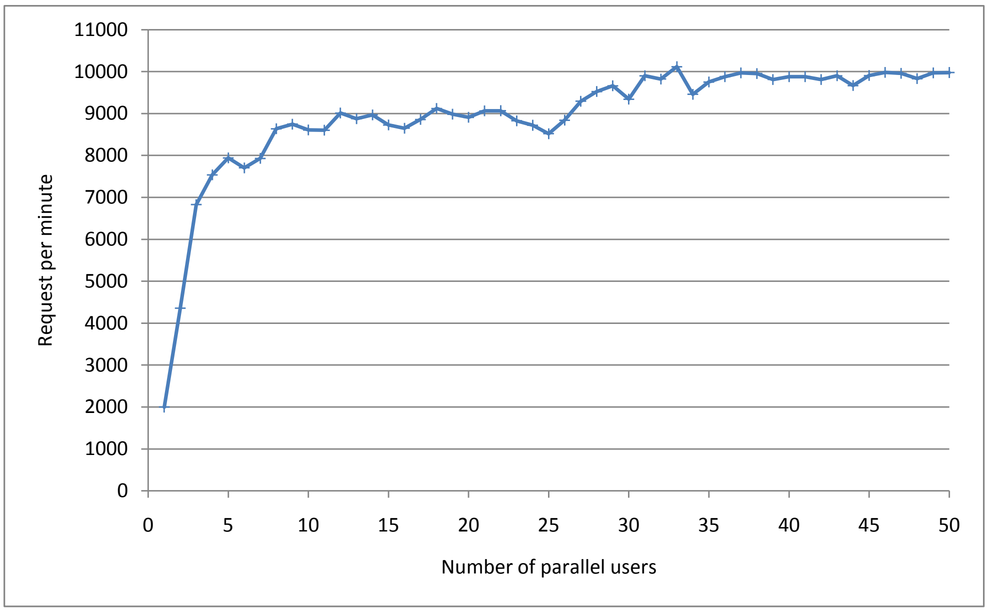


Figure 4.5: *Scalability Test* report: Average number of request per minute (throughput) at increasing levels of parallel users.

The second plot (figure 4.5) visualizes the number of requests that can be completed by the Web service in one minute. In other words, it shows the throughput of the system at increasing levels of parallel users. Of course at first the throughput can be improved by parallelizing the requests. But already beginning at a small number of 5 parallel users the throughput does not significantly grow any more in this example. The system is already used to its capacity. This test was also repeated with higher numbers of parallel users and the results show that the throughput levels off at about 10,000 successfully completed requests per minute.

Another interesting result of the *Scalability Test* is the maximum number of parallel users the system can probably serve. Using a similarly configured *Scalability Test* with more iterations it was shown that, if the number of parallel users excesses about 200, not all requests can be competed successfully.

The HTTP connections time out before responses are received and therefore some requests fail more or less randomly. Again the QoSTIL user can draw consequences from such results. E.g., if such a high number of parallel users has to be supported and if very high response times are acceptable, a simple solution would be to increase the connection timeouts on the client and server side. Of course this would only postpone the problem. Normally it is unavoidable to add extra hardware and/or optimize the Web service implementation to support even more concurrent users.

It should be noted that the throughput in this test scenario is not only determined by the system running the Web service but also heavily depends on the clients' ability to send many requests in parallel. After all, parallel users are here simulated by using threads all running on a single machine. An extension of the QoSTIL language for supporting distributed test clients is discussed in the following section 4.3.

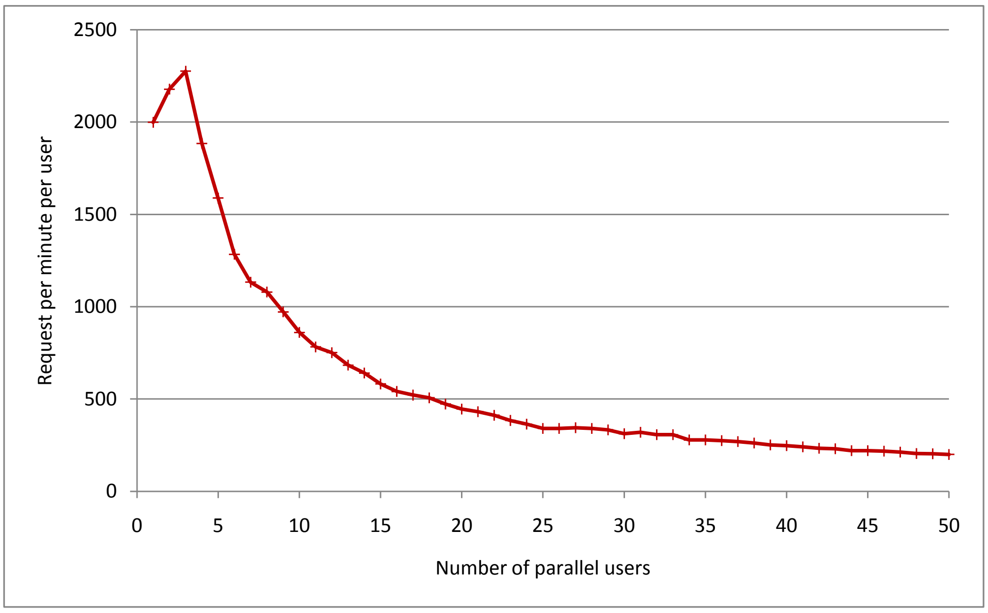


Figure 4.6: *Scalability Test* report: Average number of request per minute per user at increasing levels of parallel users.

Figure 4.6 is the last plot that was automatically generated from the data obtained by executing the *Scalability Test* for the PiApproximator Web service. It also depicts the throughput of the system at increasing levels of parallel users, but now from the viewpoint of a single user/client. It is a measurement of the performance (throughput) a user can expect if the system is under a given load. As described in section 4.1 the values are simply obtained by dividing the total throughput by the number of parallel users.

Since the total throughput in the previous plot (figure 4.5) was nearly constant for more than 5 parallel users, the curve in this plot closely resembles a simple rational function ($f(x) = \frac{a}{x}$ where a is a constant and x is the number of parallel users) for this Web service.

Nevertheless the plot can be used to read off relevant QoS attributes. For example, assume that the SLA for this service contains the condition that a single client should be able to have at least 100 requests per minute processed by the service, if there are no more than 20 clients using the service at the same time. By looking at the curve we can see that this condition is very likely to be met (assuming that the service handles all client requests in a fair way).

4.3 Possible future language extensions

As demonstrated in the previous sections, the QoSTIL in its current state can already be used to define different performance-related QoS tests for Web services and the test results can be all kind of important QoS attributes that are relevant to Web service providers and consumers. There are of course test types and QoS attribute calculations that cannot be implemented (or are difficult to implement) using the current specification of the language.

This section contains some ideas for future language extensions that are not yet specified in detail or implemented.

Distributed performance tests

The generated runtime environment for QoSTIL tests simulates multiple users accessing Web services simultaneously using threads on a single machine. Hence, the performance test results are limited by the resources of the single machine the test is executed on. For example, if the computer running the test is not able to send the desired number of parallel requests to the service under test in a timely manner, the test results will be useless. It may quite be possible that a single machine is just not able to generate enough load to stress a service to a level defined in a test specification.

Therefore in a lot of test scenarios it is desirable to run distributed performance tests. Here we define a distributed performance test as a test where the behavior of multiple Web service clients is not only simulated by threads, but by actual independent systems (machines). In addition each machine could of course still execute multiple threads. Using such a setup it becomes possible to perform real stress testing and ensure that the test results are not influenced by the way multiple clients are simulated.

Such distributed performance tests are not supported in the current implementation of the QoSTIL. There two possible ways how this feature could be added.

First, transparent distribution of parallel executions to multiple physical machines could be added to the runtime system that is generated for QoSTIL tests. The test definition and QoSTIL model does not need to be changed. When a test should be executed, the runtime system would need to be configured with the number of available test client machines. The system would then automatically distribute parallel executions to the available machines, instead of just running threads on a single machine.

Second, support for distributed executions could be added as real language elements to the QoSTIL model. The language elements could give the QoSTIL user full control how executions are to be distributed to particular test client machines. A proper syntax for this feature would of course have to be designed too. The advantage of this approach is that the test author has full control how the test should be executed. On the other hand this would mean that the test author needs to know in advance what or at least how many test client machines will be used when the test is executed. This would make a test specification less flexible and portable.

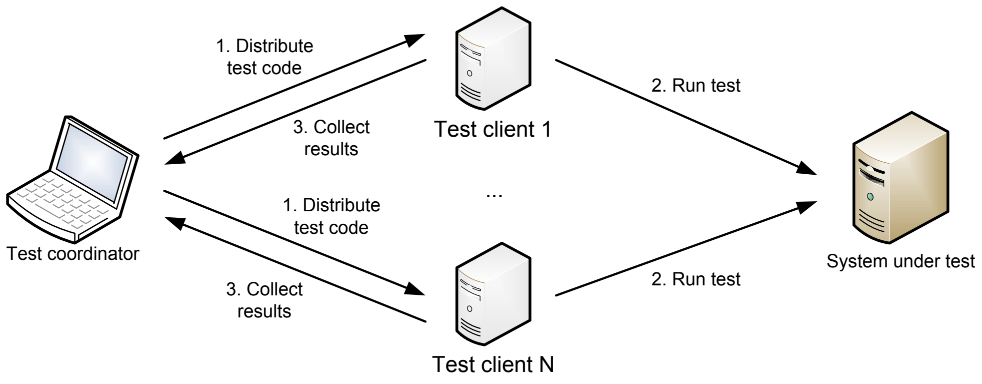


Figure 4.7: Distributed performance test architecture with test coordinator.

Either way a component that handles the distribution of the parallel executions to multiple test client machines and collects the (partial) test results from them would need to be added to the overall system. A possible design for such a system is depicted in figure 4.7. Here a test coordinator distributes the executable test code to multiple test clients. The test clients run the code to test the target system. When all clients have finished, the test coordinator can collect the test results and generate the final test report data by combining (aggregating) the results according to the QoSTIL specification.

Test termination criteria

A performance test will run as configured until it reaches some termination criterion. In the current QoSTIL model there is explicit support only for terminating after a fixed preconfigured number of iterations has been reached (given by the *count* attribute of repeated executions in a test specification).

For some test types it would be helpful to have more termination criteria: A test might be supposed to run for a defined time frame (e.g. one minute) or until some other configured limit has been reached (e.g. the maximum number of allowed errors for an assertion has been reached). An example for such a test might be a *Scalability Test* where the number of parallel clients is increased until a configured maximum acceptable response time of the Web service is exceeded. That way the maximum operating capacity of the service can be evaluated.

Aggregator plug-ins

The QoSTIL has built-in support for a basic set of aggregators (average, variance, median, minimum, maximum, sum ...) for simple processing and evaluation of sequences of test results. Of course there are many more possible aggregator functions which might be useful for particular test instances. Since the available aggregators are simply implemented by Java functions, the QoSTIL runtime system could easily be extended with aggregator plug-ins (developed as Java classes) that provide additional aggregator functions to test authors. A simple way to register such new aggregators in the language model and the syntax would need to be added to the system.

More plotting options

The current test report generation component allows visualizing performance datasets as charts. Annotated datasets will be fully automatically displayed in well-arranged colored XY line plots, without the need for any configuration. But sometimes it might be necessary to have more control over this output.

The test author might want to configure all kind of aspects of the chart output, such as legends, grids, axis titles, number formatting and units. Advanced options to control the plotting of datasets might include smoothing and interpolation.

Also different chart types should be supported in future versions. Examples are bar charts, pie charts or nets.

All these options can easily be added as parameters to the plot annotation of the QoSTIL model. Adding support in the generated runtime system is also rather effortless since the used plotting library already supports the most important options.

Automatically inferred types

When creating a test instance using the QoSTIL, a test author needs to explicitly type all field definitions (test parameters and results). During the usability evaluation it was shown, that types in the QoSTIL can at first be a little confusing. Especially report types that are automatically generated for user defined tests (and also their array variants), can interfere with the otherwise good readability of test instances for new QoSTIL users.

As an example, the user has to know that the return value of an execution of some other test, say `OtherTest`, has the type `OtherTestReport` (resp. `OtherTestReport[]` if it is a repeated execution). If the test author wants to reuse such a value directly in a test result, the test result has to be explicitly typed using this type name. It is obvious that if for the execution return value the type is already (statically) known, the type of such a test result could be inferred automatically.

Making types optional where possible would probably enhance the test instance readability for new QoSTIL users. It should be possible to infer all test result types automatically based on the (known) types of executions and test parameters.

Removing all types from the QoSTIL is not planned. We believe that explicit types at least for parameters make their semantics more obvious and the test usage less error-prone.

CHAPTER 5

Related Work

This chapter contains an overview of related work in the problem domain of performance testing of Web services. The advantages and disadvantages of other approaches are discussed in detail.

There is a lot research about performance evaluation of Web applications and Web servers in general (see e.g. [30] and [15]). Performance testing of Web services is just a specialization and most of the tools for Web server testing can also be used for simple Web service testing.

Saddik [28] wrote about performance measurements of Web services-based applications. He developed performance and scalability test cases using the TestMaker framework, provided by PushToTest [25], to evaluate the performance of an e-Learning application. The general approach to use an existing software solution for Web service performance testing is discussed in the following section 5.1.

The IBM WSLA (Web service level agreement) framework [13] can be used to specify SLAs for Web services in a formal language and automatically monitor the specified QoS guarantees. It does not focus on testing but is nevertheless interesting for the QoSTIL development since it contains a very advanced XML-based language for specifying metrics for QoS attributes.

The further discussion in this chapter will focus on two other general approaches for performance testing of Web services compared to tests developed using the QoSTIL. The first approach is the use of existing open source or proprietary performance measurement tools, and the second is the development of performance tests based on the TTCN-3.

5.1 Performance measurement tools

A popular approach for automatic performance testing of Web services is using some software testing solution. There is a multitude of testing tools available that support performance measurements for Web services. Both open source tools (e.g. Apache JMeter [34], soapUI [5]) and proprietary tools (e.g. IBM Rational Performance Tester [8], PushToTest Testmaker [25], Paessler Webserver Stress Tool [22]) allow for quick evaluation of performance-related QoS attributes using different test types, such as load tests, stress tests and scalability tests.

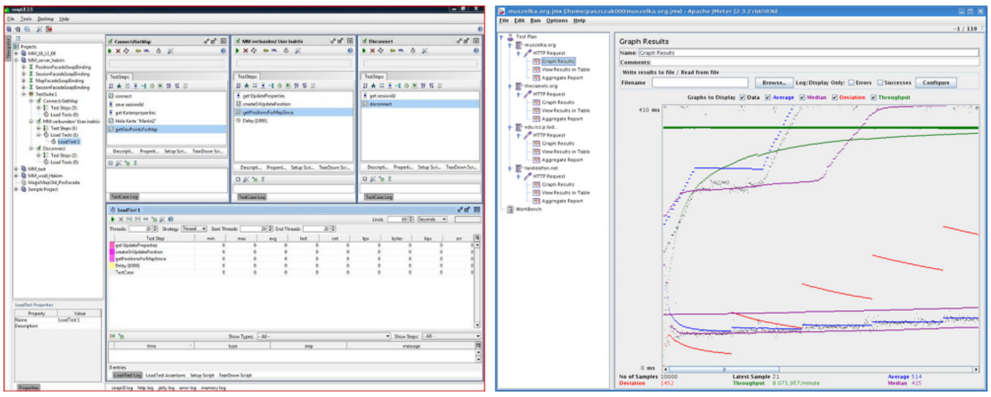


Figure 5.1: Screenshots of performance measurement tools: soapUI (left) and Apache JMeter (right).

These tools offer numerous features that faciliate to quickly assemble complex and advanced performance tests by to use configurable components. For instance, soapUI is able to simulate different predefined types of load just by selecting one in a list. This is referred to as *load strategies* by its authors. Thus, a load test can be performed using a fixed load strategy, or using a variance load strategy that varies the number of threads (simulating Web service clients) by a defined maximum value during the test, or by using another one of the other seven currently existing strategies.

A problem with these tools is that the tester is always bound to the specific features and capabilities they offer. Creating complex tests using the QoSTIL presented here might need more initial learning but offers more flexibility. For example by using the expression language offered by the QoSTIL exactly those QoS attributes that are relevant for an organization (e.g. because they are mentioned in a SLA) can be evaluated. Tests can be more complex and much more specific for a particular system that should be analyzed. On the other hand existing QoSTIL test instances can also be easily reused and extended. Hence, it is also possible to have for instance a set of prepared load strategies that can be exchanged, as with soapUI.

We believe that performance tests that are based on a clear specification rather than some process dictated by a (possibly proprietary) test tool are more valuable. Tests defined using the QoSTIL presented here are always explicitly specified as instances of the QoSTIL language model. Such a clear specification makes the test process more transparent and can ensure that the test results are objective and comparable.

Performance testing of Web services should not be done irregularly by executing some external tool, but it should be integrated with the overall development process. Also the implementation and review of the test specifications should happen early and openly. Since QoSTIL test instances are just plain text files using a defined syntax, they are self-documenting and easy to communicate. QoSTIL test instances can and should be managed together with all other software development artifacts in a revision control system. Also the test reports that are generated when a QoSTIL test is executed are openly available in the Web user interface. This advocates a constant discussion of possible performance bottlenecks of the tested system. Such an open testing process will directly and indirectly save valuable development time, because it removes the necessity for different persons to execute the same test repeatedly and it helps to identify performance problems as early as possible.

Another advantage of QoSTIL tests over performance measurement tools is the interaction of the QoSTIL with the low-level QuaLa (see section 4.1). QoSTIL tests are abstract specifications that are independent of the system that is tested and its implementation technologies. But by adding a low-level description of the system under test (using the low-level QuaLa) also implementation-specific QoS attributes can be measured. For example, the round trip time of a request can be broken up into more fine-grained intervals, such as processing and wrapping time. Performance measurement tools can normally only perform black-box testing. Since they are not aware of any Web service implementation details they can only measure outside-visible attributes (such as the complete round trip time for a request, but not the processing time on the server-side).

5.2 Performance tests based on the TTCN-3

The Testing and Test Control Notation TTCN-3 [4] is a scripting language standardized by the European Telecommunication Standards Institute (ETSI). It has been specifically designed for testing and certification of modern telecommunication and IT technologies. The TTCN-3 can be used for various kinds of tests including functional, interoperability, robustness, regression, load, and scalability testing. Complex distributed test behavior can be described in terms of sequences, alternatives, and loops of stimuli and responses. The test system can use a number of test components to perform test procedures in parallel.

Schieferdecker et al. [29] have developed a test framework for Web services based on the TTCN-3. They apply the TTCN-3 for system-level tests that check how a system performs for single service requests and scales as the number of service requests using it increases. Similar to the approach described in this thesis, they have also developed a hierarchy of tests for Web services. Complex load and scalability tests are defined by reusing basic functional tests for the system under test. These predefined test scenarios and test setups can be adapted to different systems under test by exchanging the modules for the basic functional tests only.

A major advantage of applying TTCN-3 for implementing Web service performance tests is its support for fully distributed tests. For instance, Schieferdecker et al. have deployed their load test for Web services on three test containers (standard PCs) and equally distributed the test components on each of them. Distributed tests are currently not supported in the QoSTIL, but support is planned for future versions (see section 4.3).

Our decision to define our own new performance test language for Web services and not just apply the existing TTCN-3 is based on the belief that a Web service performance test language should be simple and domain-specific. The TTCN-3 is applicable to a wide range of tests for various technologies. It was originally developed for testing the conformance and interoperability of communication protocols. This has resulted in a complex syntax with many different concepts that have to be learned before they can be applied. Actually TTCN-3 could be described as a general purpose programming language (it supports all typical constructs, such as conditions and loops) with a lot of additional concepts that ease the definition of all kinds of tests. The QoSTIL on the other hand has a clear language model that focuses on the instrumentation and evaluation of performance tests. By integrating the low-level QuaLa support for Web services is added. The QoSTIL was developed to be a helpful tool for a very particular purpose (Web service performance testing).

An advantage of the test system presented in this thesis is the generation of test reports in the Web user interface. To our knowledge the TTCN-3 based framework by Schieferdecker et al. does not consider the automatic visualization of test results.

Moreover, the existing work on performance tests based on TTCN-3 focuses only on black-box tests for Web services. Contrary to QoSTIL tests, it is not possible to evaluate implementation-specific QoS attributes, such as the wrapping or processing time on the server-side.

CHAPTER 6

Summary and Conclusions

In this thesis a new domain-specific language, named QoSTIL, for Web service performance testing is introduced. It is designed to describe Web service client workload and the measurement and evaluation of performance-related quality attributes of Web services, such as response times, throughput and scalability. Test instances can be automatically transformed to executable Java code, which is ready to run the defined tests for a specific target system. The developed test runtime system also includes a database for storing test results, and a Web user interface for visually presenting test reports.

The provided language and performance tests written using it are generic as they can be used for arbitrary Web services and do not depend on specific implementation technologies (e.g. Web services frameworks). Only by combining test instances with a low-Level QuaLa [18, 36] description of the system under test, the specifics of concrete Web services and the relevant implementation-specific measurements become available.

The QoSTIL is developed by following the MDD paradigm. The defined textual DSL syntax is just a notation for a precisely specified language model. The central element of the language model is a *Test*, which consists out of

Parameters, *Executions* (describing the behavior of the test), and *Results* (together with an expression specifying the result value). The external syntax for notating such tests has been designed to look familiar and intuitive for developers knowing Java (or other C-like languages).

Tests can be composed to create more complex tests and complete test plans. This is achieved by language constructs to (repeatedly) execute tests inside tests. Multiple tests can also be executed in parallel, which is necessary to simulate the behavior of several concurrent Web service clients.

In the language, *Aggregators* can be used to summarize the results of repeatedly executed tests and perform statistical data analysis. This allows the flexible measurement and evaluation of business-specific quality attributes.

The practical applicability of the QoSTIL is demonstrated by a hierarchy of Web service performance tests that was developed using it. At the bottom there is the simplest test that requests a service once and evaluates basic performance attributes, such as processing, latency, wrapping and roundtrip time. The most complex sample test is a *Scalability Test*, that evaluates the scalability of a Web service by observing its throughput at increasing levels of concurrently running clients.

All developed tests are executed in a demonstration environment for a representative Web service. By discussing the report of an execution of the *Scalability Test* in detail, it was shown that the results are valuable and significant, and that the language is ready for the task it was developed for.

While the defined language can already be practically used to comprehensively test Web service performance, possible future language extensions were also discussed in the work, which could make it even more expressive or simpler to use. These include extensions for distributed tests, more test result plotting options and automatic type inference.

In conclusion, it can be said that the developed QoSTIL has some considerable advantages over existing approaches for Web service performance testing: It is a rather simple language based on a precisely specified language model that was specifically tailored for the task of Web service performance testing. Test instances are plain text files and are therefore easy to communicate and maintain. They are reusable, self-documented and advocate a transparent and early testing process. After executing the tests, reports are stored in a central place and are openly accessible through a Web user interface. Those reports are well-structured, the results are clearly visualized and they contain all the information that is necessary to reproduce a performed test. Last but not least, the language is not only capable of black-box testing: By integrating a low-level description of the system under test, it does also support implementation-specific measurements at the client and server-side.

Acknowledgement

I would like to thank my advisors Uwe Zdun and Ernst Oberortner for sharing their expertise and helping me during the realization of this work.

I heartily thank my parents, Jutta and Georg, who always supported me during my whole studies. Special thanks go to my sisters, Angelika and Isabella, for taking the time to proofread this thesis.

Finally, I kiss my girlfriend for always encouraging me to question things and her creative advice for the thesis presentation poster.

List of Figures

List of Tables

Code Listings

Bibliography

[1] Java Compiler Compiler (JavaCC). `https://javacc.dev.java.net/`. Last accessed: Oct 8, 2010.

[2] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *Proceedings of the 4th international workshop on Software and performance*, WOSP '04, pages 94–103, New York, NY, USA, 2004. ACM.

[3] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[4] ETSI Methods for Testing and Specification (MTS). *The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, 2010. ES 201 873-1, Latest Version: 4.2.1.

[5] eviware. soapUI. `http://www.soapui.org/`. Last accessed: Feb 2, 2011.

[6] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? `http://www.martinfowler.com/articles/languageWorkbench.html`, May 2005. Last accessed: Jan 4, 2011.

[7] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[8] IBM. Rational Performance Tester. `http://www.ibm.com/software/awdtools/tester/performance/`. Last accessed: Feb 2, 2011.

[9] Java Community Process. JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.0. `http://jcp.org/en/jsr/detail?id=224`. Last accessed: Oct 8, 2010.

[10] Java Community Process. JSR 314: JavaServer Faces Technology. `http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html`. Last accessed: Oct 8, 2010.

[11] JBoss Inc. The Seam Framework. `http://seamframework.org/`. Last accessed: Oct 8, 2010.

[12] Li jie Jin, Vijay Machiraju, and Akhil Sahai. Analysis on Service Level Agreement of Web Services. Technical report, HP Laboratories, 2002.

[13] Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[14] J. McGovern, S. Tyagi, M. Stevens, and S. Matthew. *Java Web Services Architecture*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.

[15] Daniel A. Menascé. Load Testing of Web Sites. *IEEE Internet Computing*, 6:70–74, 2002.

[16] Daniel A. Menascé and Virgilio Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[17] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. Domain-specific languages for service-oriented architectures: An explorative study. In *Towards a Service-Based Internet, First European Conference, ServiceWave 2008, Proceedings*, pages 159–170, Vienna, Austria, December 2008. LNCS 5377, Springer-Verlag.

[18] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. Tailoring a model-driven quality-of-service dsl for various stakeholders. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, MISE '09, pages 20–25, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. Patterns for measuring performance-related qos properties in distributed systems. In *17th Conference on Pattern Languages of Programs*, October 2010.

[20] Liam O'Brien, Paulo Merson, and Len Bass. Quality attributes for service-oriented architectures. In *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Ole Laursen. flot - Attractive Javascript plotting for jQuery. `http://code.google.com/p/flot/`. Last accessed: Oct 8, 2010.

[22] Paessler AG. Webserver Stress Tool. `http://www.paessler.com/webstress`. Last accessed: Feb 2, 2011.

[23] Michael P. Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, 2007.

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

[25] PushToTest. TestMaker. `http://www.pushtotest.com/`. Last accessed: Feb 4, 2011.

[26] Shuping Ran. A Model for Web Services Discovery with QoS. *SIGecom Exch.*, 4(1):1–10, 2003.

[27] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 205–212, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Abdulmotaleb El Saddik. Performance Measurements of Web Services-Based Applications. *IEEE Transactions on Instrumentation and Measurement*, 55:1599 – 1605, 10 2006.

[29] Ina Schieferdecker, George Din, and Dimitrios Apostolidis. Distributed functional and load tests for Web services. *International Journal*

on Software Tools for Technology Transfer (STTT), 7:351–360, 2005. 10.1007/s10009-004-0165-6.

[30] J. Shaw. Web Application Performance Testing - a Case Study of an On-line Learning Application. *BT Technology Journal*, 18:79–86, April 2000.

[31] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[32] The Apache Software Foundation. Apache Ant. `http://ant.apache.org/`. Last accessed: Oct 8, 2010.

[33] The Apache Software Foundation. Apache CXF: An Open Source Service Framework. `http://cxf.apache.org/`. Last accessed: Oct 8, 2010.

[34] The Apache Software Foundation. Apache JMeter. `http://jakarta.apache.org/jmeter/`. Last accessed: Feb 2, 2011.

[35] Juha-Pekka Tolvanen. Domain-Specific Modeling: How to Start Defining Your Own Language. `http://www.devx.com/enterprise/Article/30550`, 2008. Last accessed: Oct 8, 2010.

[36] Huy Tran, Ta'id Holmes, Ernst Oberortner, Emmanuel Mulo, Agnieszka Betkowska Cavalcante, Jacek Serafinski, Marek Tluczek, Aliaksandr Birukou, Florian Daniel, Patricia Silveira, Uwe Zdun, and Schahram Dustdar. An end-to-end framework for business compliance in process-driven soas. *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on*, 0:407–414, 2010.

[37] W3C. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, 2001.

[38] W3C. SOAP Version 1.2. `http://www.w3.org/TR/2007/REC-soap12-part0-20070427/`, 2007.

[39] Narada Wickramage and Sanjiva Weerawarana. A Benchmark for Web Service Frameworks. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 233–242, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Uwe Zdun. The Frag Language. `http://frag.sourceforge.net/`. Last accessed: Oct 8, 2010.

[41] Uwe Zdun. A DSL toolkit for deferring architectural decisions in DSL-based software design. *Inf. Softw. Technol.*, 52:733–748, July 2010.