Integrating Performance Tests in a Generative Software Development Platform

Diploma Thesis

Eike Schulz

June 2, 2014

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Examiner:	Prof. Dr. Wilhelm Hasselbring
Advisor:	Dipl.-Inform. André van Hoorn (U Stuttgart)
	Dr. Wolfgang Goerigk (b+m Informatik AG)

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Performance is a business-critical quality factor particularly for e-commerce companies that operate platforms with thousands or millions of users. Any dissatisfaction of customers, e.g., caused by long response times or low throughput, might end up in huge financial losses. Consequently, e-commerce platforms must hold a tolerable level of overall performance to ensure a positive user experience. In times of Web 2.0, this demand becomes even more difficult to meet, since complexity of Web systems increases rapidly; on the other hand, server capacity planning as well as scalability evaluation require reliable methodologies for any type of performance testing.

Many performance testing tools like the widely used JMeter build on the concept of test scripts. Therewith, these tools offer benefits like test repeatability and consistency. In contrast, the manual specification of those scripts is an error-prone and time consuming issue. Hence, companies tend to limit their scope of performance testing, to keep the investment costs for such tests low. To overcome this problem, alternative, cost-saving approaches are required. One desirable aim is to retrieve test scripts straight from the model-driven software generation process of a system. Corresponding applications often depend on workflow definitions, which specify valid user activities with respect to system usage. These definitions indicate the user behavior accepted by the related system. Therewith, they provide significant information for testing models which build on emulation of user behavior. On the other hand, such models need to be parameterized with input values which might be retrieved from monitoring data.

Utilization of workflow information for generating testing models and parametrization of such models with monitoring data are base tasks of an approach which will be followed in this thesis. The approach builds on the workload generation model of the Markov4JMeter add-on for JMeter, which itself builds on probabilistic user behavior. For the sake of clarity, the approach has been divided into three tasks. The first task includes the generation of so-called Test Plans for the JMeter tool from a Markov4JMeter workload model. For this purpose, a domain-specific language (DSL), namely M4J-DSL, will be defined. M4J-DSL denotes a meta-model for Markov4JMeter workload models, and model instances provide all information required for the transformation to a corresponding Test Plan – not limited to JMeter. The transformation itself is performed by a framework which has been developed for this purpose. The second task aims to the derivation of an appropriate M4J-DSL model from so-called Screenflows of a concrete, generative software development platform, namely b+m gear. That task requires user behavior information, such as think times, which can be retrieved from monitoring data. The extraction of such data from user session traces will be discussed in the third part of the approach. That part includes an implementation of think time support in the Markov4JMeter add-on. Though mainly aiming to performance testing, the underlying ideas of the introduced approach might be used for alternative testing types, such as regression tests. This work identifies challenges and open issues in context of test generation, which mainly result from the complexity of AJAX systems and mismatches between workload models and underlying models of generated applications.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

AJAX	Asynchronous JavaScript and XML
AL	Apache License
API	application programming interface
AST	abstract syntax tree
BPMN	Business Process Modeling Notation
BSD	Berkeley Software Distribution
CPU	central processing unit
CSS	Cascading Style Sheets
CSV	comma-separated values
DFS	depth-first search
DSL	domain-specific language
EFSM	extended finite state machine
EMF	Eclipse Modeling Framework
EPL	Eclipse Public License
LGPL	Lesser General Public License
GUI	graphical user interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
HVI	high value interface
IDE	integrated development environment
JAR	Java Archive
JPA	Java Persistence API
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
M2M	model-to-model
M4J-DSL	Markov4JMeter Domain-Specific Language
MDD	model-driven development
MDSD	model-driven software development
MDSE	model-driven software engineering
OCL	Object Constraint Language
PS	production system
QoS	quality of service
RPC	Remote Procedure Call
SOA	service-oriented architecture

SOAP	Simple Object Access Protocol
SUT	system under test
TCP	Transmission Control Protocol
UI	user interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

With the growing influence of network technology and e-commerce, *performance* has been identified as a business-critical factor. Customers generally interpret performance characteristics like response times or throughput as an indicator for the quality of service (QoS), and they are sensitive to any technical matters which prevent them from completing their business activities in a timely manner [Menascé et al., 1999]. For companies that operate platforms with thousands of users, the dissatisfaction of even a fractional number of customers might end up in huge financial losses. Hence, the overall performance of a system must be kept at a tolerable level. The system capacity must be adjusted to the actual system usage, to avoid cost explosions through over-provisioning or user dissatisfaction through under-provisioning. Distributed systems in particular must be scaled according to a satisfactory performance. Consequently, server capacity planning and scalability require *reliable* methodologies for any type of performance testing.

Performance testing tools like *JMeter* [The Apache Software Foundation, 2013] support test automation based on the concept of test scripts, which include predefined request sequences for being replayed when the considered system is under test. Therewith, these tools offer benefits like repeatability and optimization [Dustin et al., 1999, p.41]. In contrast, the manual specification of such request sequences, including protocol-specific parameters, is often an error-prone and time consuming issue; for any changes in the targeted system, the related scripts need to be adjusted, which might lead to cost overruns through an increased effort [Dustin et al., 1999, p.108]. Model-based testing holds a particular challenge due to the nature of model-based tests: the manual construction of valid tests requires the specification of input parameters, e.g., probability values, which must be mostly estimated by the tester; the estimation of such values is a difficult task [Schulz, 2013]. Another challenge is the construction of performance tests for systems which have "black box" character to the tester: if the system specification is not available, requests need to be recorded at first for being analyzed in terms of the transferred data; tools like JMeter provide proxy server functionality for those tasks in context of Web-based systems, but nevertheless it requires an enormous effort to identify the key information which is needed for constructing appropriate test scripts [Schulz, 2013].

The drawbacks of hand-crafted performance tests awake the desire to generate tests, based on information which might be already available from the generation process of a system to be targeted. In particular, generated applications depend on models which in many cases include workflow definitions, from which valid user action sequences can be retrieved. Such definitions might be used for deriving test scripts which send requests according to expected user behavior. Appropriate test input parameters which represent probabilistic user behavior might be achieved via monitoring techniques.

This thesis aims to the integration of performance tests into a generative software development platform which builds on models for the applications it generates. It introduces an approach for deriving performance tests from models and discusses the information which must be provided by these models. Though mainly aiming to performance testing, the underlying ideas of the introduced approach might be even used for alternative testing types, such as regression tests. Section 1.2 illustrates the goals and scope of this thesis, and Section 1.3 gives an overview of the implementation approach followed in this work.

1.2 Scope and Goals

In context of the research project *DynaMod* [van Hoorn et al., 2011, 2013], which aimed to the semi-automatic modernization of legacy software systems, a model-driven load testing approach has been evolved for generating probabilistic workload, based on user traces. The approach builds on a probabilistic workload generation model that uses Markov chains for describing the behavioral patterns of different user types. The model has been developed by van Hoorn et al. [2008] for their JMeter add-on *Markov4JMeter* [Software Engineering Group – Kiel University, 2013], which aims to the emulation of real user behavior. Markov4JMeter has been used for applying the evolved approach on a model-driven, Web-based case study system, shortly referred to as *system under test (SUT)*. The research findings confirmed several expected challenges, and moreover, possible solutions for some of these challenges have been identified [Schulz, 2013; Schulz et al., 2014]:

1. *Trial-and-error approach.* In case the SUT is a "black box" for the tester, that is no protocol-specific information is available, a trial-and-error approach is needed to gain the required information. For Web-based systems, a proxy server can be used for those purposes.

   Several performance tools facilitate such tasks, e.g., JMeter provides a proxy server component which is of great help, as well as the drag-and-drop functionality of JMeter's graphical user interface (GUI).

2. *Estimation of probabilities.* Probabilistic input parameters of a workload model are generally difficult to be estimated, and the validity of a resulting test script is doubtful. The workload model used in the approach has input parameters which must be estimated with regard to the actual user behavior. Those values can be gained from user traces alternatively, but such traces are sometimes not available.

This issue can be solved through extracting model input values from user traces, which have been monitored on an alternative (production) system. This could be, for example, a legacy system with similar functionality like the SUT.

3. *Coverage of scenarios.* The effort for covering all possible system usage scenarios is high, since users might execute services in almost arbitrary order, infinitely often. The number of execution variations increases rapidly for even a small amount of services.

   As a solution, user traces recorded in the past can be used for identifying possible scenarios. Additionally to service execution sequences, those traces might even provide information such as time stamps for user think times extraction.

4. *Fragility of test scripts.* Test scripts, which include protocol-specific information like Uniform Resource Identifiers (URIs) or parameter names, are fragile with respect to protocol changes. In worst case scenarios, a complete script must be updated, even if only small modifications have been made on the related SUT.

   For MDSD applications, a desirable solution is to generate script templates with included protocol data, derived straight from the models of the generated application. Those templates could be enriched with remaining model parameters, instead of struggling with the complete set of protocol-specific information.

5. *Validity of load test scripts.* The validity of load test scripts is generally difficult to be determined. One option would be a measurement-based evaluation by means of a system which is in productive operation.

For the first two items, solutions have been already given; corresponding techniques have been used in the DynaMod project. However, both issues require a lot of manual work, slowing down the whole configuration process for a load test. Hence, the automation of relevant tasks is particularly desirable to improve the efficiency as well as the reliability of the testing process. Item 5) still constitutes an open issue for future work. Items 3) and 4) indicate the main goals of this work, involving the automation of items 1) and 2). This will be discussed in the next Chapter.

1.3 Overview of the Implementation Approach

Figure 1.1 gives an overview of the implementation approach followed in this thesis. It depicts the considered SUT, which constitutes an application generated through a platform named *b+m gear* [b+m Informatik AG, 2013]. Both systems will be introduced in Section 2.5. The dotted frames indicate the main components of the approach and therewith the implementation chapters of this thesis, which are indicated in braces. The following section gives a more detailed overview of the document structure.

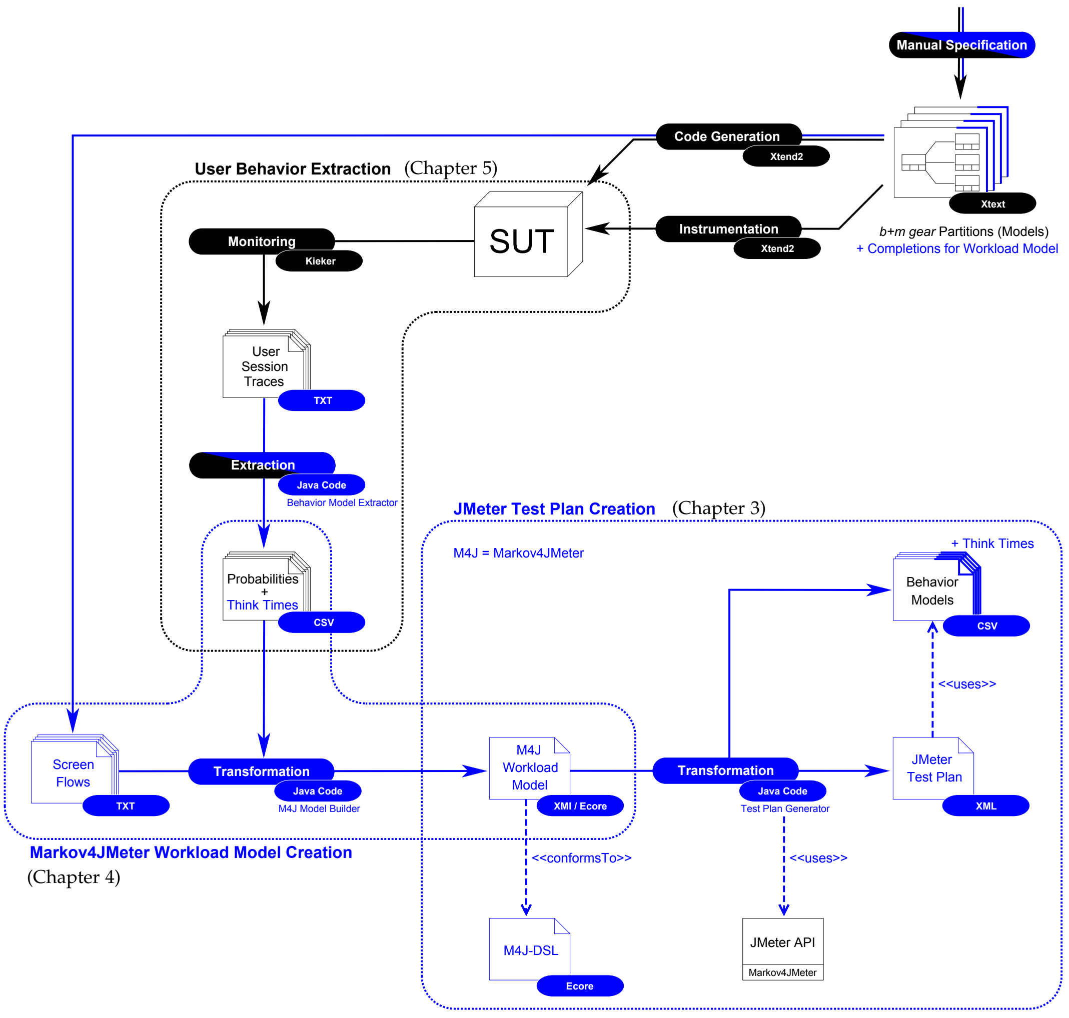


Figure 1.1. Overview of the implementation approach, illustrating the three main components. Black-colored parts indicate already existing artifacts and proven functionality respectively, blue-colored parts indicate implementation tasks of this work.

1.4 Document Structure

The remainder of this document is structured as follows:

▻ Chapter 2 provides the foundations and used technologies.

▻ Chapter 3 presents a DSL for Markov4JMeter, namely M4J-DSL, and illustrates the transformation from M4J-DSL models to JMeter Test Plans, including an introduction of the related Test Plan Generation Framework.

- ▻ Chapter 4 outlines the transformation from b+m gear Screenflow information to M4J-DSL workload models, including an introduction of the related M4J-DSL Model Generator Framework. Additionally, it introduces a dedicated Java Sampler which builds on Java Reflection techniques.
- ▻ Chapter 5 describes the extraction of user behavior information from user session traces that have been monitored on a given SUT. It also describes the think time emulation support which has been added to the Markov4JMeter add-on.
- ▻ Chapter 6 includes the evaluation.
- ▻ Chapter 7 presents related work in context of test generation.
- ▻ Chapter 8 draws the conclusion and gives an outlook to future work.

Chapter 2

Foundations and Technologies

This chapter gives an introduction to the terms and concepts used in this thesis. Section 2.1 briefly outlines the idea of performance testing, including a classification into different testing types. It explains the meaning of model-based testing with regard to analytic workload generation. Section 2.2 describes the probabilistic workload generation model, which is the basis for the Markov4JMeter add-on for the performance testing tool JMeter. Section 2.3 gives a further insight into the JMeter tool, including an introduction to the concept of Test Plans used by JMeter. The main ideas of model-driven software development (MDSD) will be discussed in Section 2.4, and Section 2.5 gives a description of the generative software development platform b+m gear which will be considered in this thesis, including an introduction to the case study system CarShare. Section 2.8 illustrates the model-driven performance testing approach which has been developed in the context of the DynaMod project.

2.1 Model-Based Performance Testing

Performance testing can be defined as putting a system under test (SUT) under certain synthetic workload in a controlled environment, aiming to the validation of resource utilization, responsiveness and stability of the targeted system [Abbors et al., 2012]. Thereby, *synthetic workload* denotes the workload which is generated by a dedicated generation tool, shortly referred to as *workload generator*. A workload generator can run thousands of threads, with each thread sending requests which could result from a real user. Hence, these threads are shortly referred to as *virtual users*.

Based on their individual goals, performance tests can be classified into different types. Subraya and Subrahmanya [2000] distinguish between the following three types:

▻ *Load tests* are used for testing the usability of a system under day-to-day conditions. In such tests, the synthetic workload must correspond to workload which could result from a certain population of real users. Barford and Crovella [1998] denote such synthetic workload as being *representative*. Generating representative workload is one of the core challenges in load testing, since virtual users must follow the behavioral patterns of real users. Furthermore, the *think times* [Menascé et al., 1999] which denote the time between two requests sent by a user, must be realistic.

▻ *Stress tests* are used for identifying so-called "bottlenecks", that is components which might reduce the overall system performance. A stress test corresponds to a load test with think times being removed [Subraya and Subrahmanya, 2000].

▻ *Endurance tests* are used for testing the durability of a system. They correspond to load tests or stress tests with a much longer duration period [Subraya and Subrahmanya, 2000].

In this thesis, focus will be put on load testing, since the underlying ideas of load tests can even be applied to other performance testing types. For the generation of workload, Barford and Crovella [1998] distinguish between *trace-based* and *analytical* approaches, describing them as follows:

▻ A trace-based approach builds on prerecorded usage data from real users, e.g., server logs or monitoring data. The generated workload corresponds to the sequence of recorded requests. This approach can be easily implemented, but it gives no further insight into particular workload characteristics and their influence on performance results, making the identification of reasons for certain system behavior difficult.

▻ An analytical approach uses mathematical models for workload generation and does not have the drawbacks of a trace-based approach. However, more effort is required to identify the characteristics to be modeled, to measure these characteristics, and to combine the results to a single output.

This thesis builds on an analytical model which uses Markov chains for modeling probabilistic user behavior. The model is a fundamental part of the Markov4JMeter add-on for JMeter and will be described in the following section, including a short introduction to Markov chains.

2.2 The Markov4JMeter Workload Generation Model

The Markov4JMeter add-on for JMeter has been developed by van Hoorn et al. [2008] and aims to the generation of probabilistic workload for session-based systems. A *session* is defined as a sequence of service invocations done by the same user, whereas a *service* can be considered as a use case, e.g., signing in to a website, or adding an item to the shopping cart of a Web store [van Hoorn et al., 2008]. For modeling probabilistic user behavior on service-level, the Markov4JMeter add-on uses an analytical model which consists of the following four components:

▻ An *Application Model*, consisting of two layers, namely *Session Layer* and *Protocol Layer*.

The Session Layer defines the allowed sequences of service invocations, using an extended finite state machine (EFSM) whose states are associated with the services provided by the related system. A transition from a state s_1 to a state s_2, that is a transition between the associated services, indicates that service “s_2” is allowed to be invoked after service “s_1”. Transitions might be labeled with guards and actions, allowing to model transitions which fire under certain conditions only. States and transitions of the Session Layer are also shortly referred to as *Application States* and *Application Transitions* respectively.

The Protocol Layer defines for each service the related protocol-specific workflow of the system. Therefore, it contains a corresponding EFSM for each state of the Session Layer. The states of these EFSMs, namely *Protocol States*, are associated with protocol-specific information of the related system. A transition from a Protocol State p_1 to p_2 indicates that the related system needs to process the protocol-specific information in the given combination for performing the assigned service.

▷ A set of *Behavior Models*, defining the probabilistic behavior of user types to be emulated. A *user type* describes a set of users with similar behavioral patterns, e.g., occasional buyers or heavy buyers in a Web store [Menascé et al., 1999]. Each Behavior Model corresponds to a Markov chain with services as states and transitions labeled with values, which specify the probability of the related target service being invoked next by the user. Hence, states of a Behavior Model are also called *Markov States*. Figure 2.1 shows a Markov chain of a Behavior Model which is defined on states 0.01 *login*, 1.01 *add item*, 1.02 *edit item*, 1.03 *delete item*, and a dedicated exit state. *Think times*, which are defined by Menascé et al. [1999] as the average time between a user’s completed request on server side and the arrival of the user’s next request, are assigned to transitions, too [Menascé et al., 1999]. They have been not supported by the Markov4JMeter add-on in earlier versions and constitute an implementation issue solved in context of this thesis. This will be discussed in Section 5.2

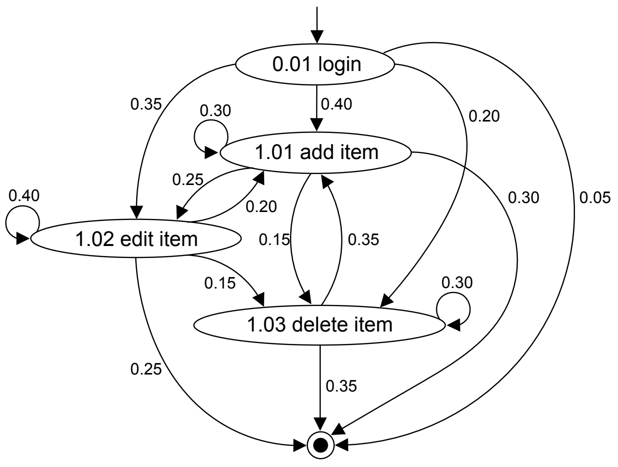


Figure 2.1. Markov chain of a Behavior Model, including four states named *0.01 login*, *1.01 add item*, *1.02 edit item*, *1.03 delete item*, and a dedicated exit state. Transitions are labeled with probabilities.

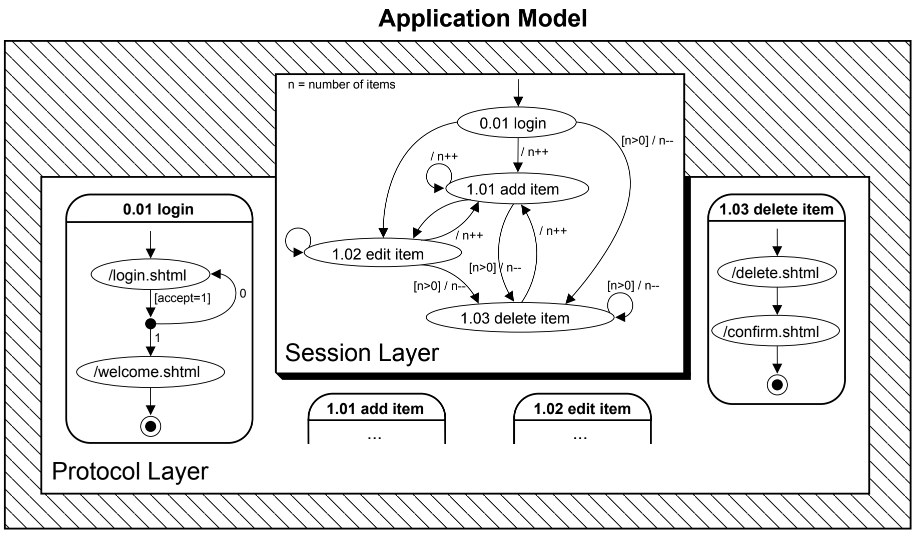


Figure 2.2. Example of an Application Model including Session Layer and Protocol Layer. The EFSMs for the services *1.01 add item* and *1.02 edit item* have been omitted for the sake of clarity, as well as the dedicated exit state of the Session Layer including its ingoing transitions.

▷ A *Behavior Mix*, specifying the relative frequency of each user type respectively its associated Behavior Model to occur.

▷ A *workload intensity*, indicating the (possibly varying) number of virtual users to be emulated in a test run. It is defined by a formula which specifies the number of users in dependency of the experiment time.

Figure 2.2 shows an example of an Application Model for a system which provides services for signing in, as well as for adding, editing, and deleting items, as indicated by Figure 2.1. The Session Layer contains the EFSM which defines the allowed sequences of service invocations, e.g., a user must be signed in before any item might be modified. The guards and actions of the transition labels regulate the counting of the current number of items and correlate to each other. It is important to note that an action is executed prior to the target service of a transition. Hence, it is assumed in the example that the services for adding or deleting an item do not provide an operation for cancellation, to keep the current number n of items consistent.

2.3 JMeter

The performance testing tool *JMeter* is a Java-based application which is available under the Apache License 2.0. The tool is able to perform any type of performance tests, and it supports miscellaneous protocols, e.g., HyperText Transfer Protocol (HTTP), HyperText Transfer Protocol Secure (HTTPS), Simple Object Access Protocol (SOAP), or Transmission Control Protocol (TCP). JMeter provides a GUI for facilitating the configuration of performance tests as well as the analysis of test results. Furthermore, it provides a documented application programming interface (API) for extensibility purposes and development of customized add-ons as well.

Section 2.3.1 discusses the configuration of performance tests via JMeter and introduces the concept of so-called Test Plans. Section 2.3.2 gives an insight into the use of the *Markov4JMeter* add-on for JMeter, which aims to the generation of probabilistic workload.

2.3.1 Test Plans in JMeter

Performance testing in JMeter is based on so-called *Test Plans*. A Test Plan defines the activities to be performed by JMeter in an experiment. In general, it is organized as a hierarchical tree structure, which consists of elements for miscellaneous purposes, e.g., for logic control, sending HTTP requests, or (result) data processing. The JMeter GUI provides drag-and-drop technology for facilitating the construction of Test Plans which are stored as *Extensible Markup Language (XML)* formatted files. Storing Test Plans in XML format also allows their construction and modification via text editor, apart from the JMeter GUI as well. This is an important aspect for development issues, since it simplifies the analysis of Test Plans with standard utilities. JMeter also provides a Java-API, which allows to construct Test Plans via programming. Figure 2.3 shows the JMeter user interface (UI) with an example of a Test Plan which contains a subset of available elements like *Timers* and *Request Samplers*.

JMeter can emulate an arbitrary (possibly varying) number of virtual users, grouped into so-called *Thread Groups*. This name results from the fact that each virtual user is emulated as an individual thread. All virtual users of a Thread Group process the same Test Plan. Hence, the definition of the related Test Plan needs to include the behavioral patterns of all user types to be emulated.

2.3.2 Markov4JMeter Add-On for JMeter

The *Markov4JMeter* add-on for JMeter extends the performance testing tool's core functionality, aiming for probabilistic workload generation. It supplies additional Test Plan elements, namely *Markov State* and *Markov Controller*, for emulating users whose behavioral patterns can be described with the workload generation model introduced in Section 2.2. The add-on particularly provides the installation of Behavior Models, Behavior Mix, and workload intensity into Test Plans.

Each Behavior Model is stored as a transition probability matrix over a common set of states in a comma-separated values (CSV) formatted file. The Test Plan elements of the Markov4JMeter add-on correlate with these files as follows:

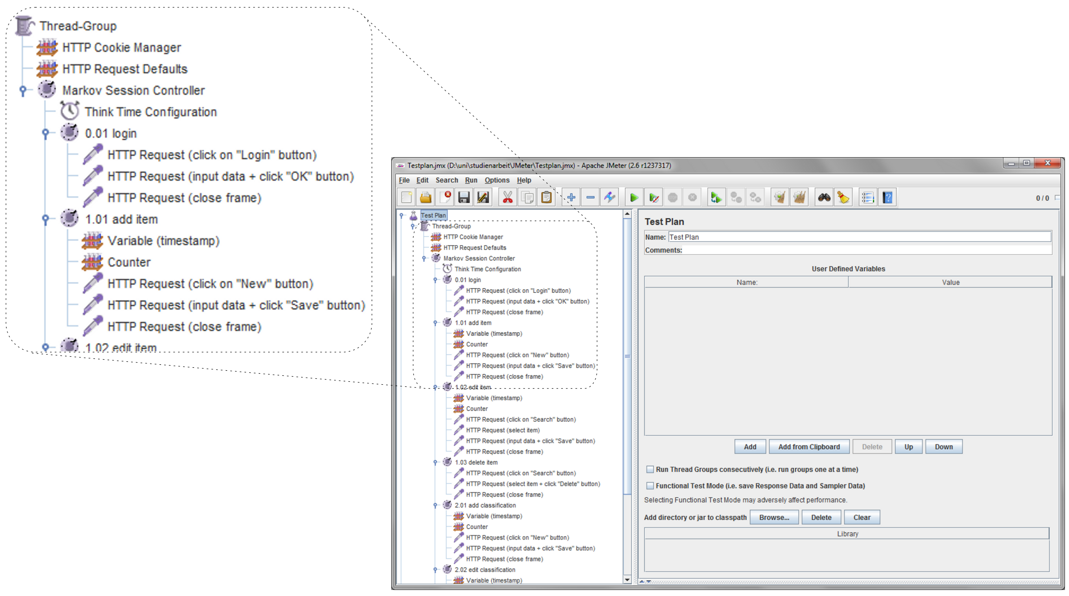

Thread-Group
HTTP Cookie Manager
HTTP Request Defaults
Markov Session Controller
Think Time Configuration
0.01 login
HTTP Request (click on "Login" button)
HTTP Request (input data + click "OK" button)
HTTP Request (close frame)
1.01 add item
Variable (timestamp)
Counter
HTTP Request (click on "New" button)
HTTP Request (input data + click "Save" button)
HTTP Request (close frame)
1.02 edit item
Test Plan
Name: Test Plan
Comments:
User Defined Variables
Add
Add from Clipboard
Delete
Up
Down
Run Thread Groups consecutively (i.e. run groups one at a time)
Functional Test Mode (i.e. save Response Data and Sampler Data)
Add directory or jar to classpath
Browse...
Clear
Library

Figure 2.3. JMeter UI with an example Test Plan [Schulz, 2013]

▻ A Markov State represents a state of a Behavior Model, with being identified by its name. Protocol-specific information can be added to the state in the form of JMeter Samplers and controllers as child nodes. Each Markov State allows the definition of its outgoing transitions in accordance to the Session Layer, including guards and actions. For example, a boolean guard expression which is always evaluated to false indicates an invalid transition.

▻ A Markov Session Controller provides input options for the Behavior Mix, that is the relative frequencies of Behavior Models. Furthermore, it provides the generation of a template CSV file with an empty transition probability matrix over all Markov States defined in a Test Plan. The template can be filled with probability values afterward for describing a corresponding Behavior Model. As a sub-component of the Markov Session Controller, the *Session Arrival Controller* allows the definition of workload intensity as a formula in dependency of the experiment time.

Both Test Plan elements provide the additional session- and protocol-specific information which is required for modeling the Application Layer of the previously described workload model. Distribution of think times is not supported by the Markov4JMeter add-on yet and constitutes one of the goals of this thesis.

2.4 Model-Driven Software Development

This section gives a short introduction to model-driven software development (MDSD). Section 2.4.1 discusses several terminology issues and gives an overview of the model-driven software engineering (MDSE) methodology. Section 2.4.2 outlines the specification of Ecore models in Eclipse, illustrates how those models can be enriched with Object Constraint Language (OCL) constraints, and closes with a short introduction to the Xtext framework.

2.4.1 Terminology

Model-driven software development (MDSD) is a variant of the *model-driven development (MDD)* paradigm, focusing on the development of software and using models as primary artifacts in a development process; it can be seen as a subset of *model-driven software engineering (MDSE)*, which on the other hand is defined as a methodology for using the benefits of modeling in context of software engineering activities [Brambilla et al., 2012, p.9]. Figure 2.4 gives an overview of the MDSE methodology. Brambilla et al. [2012, p.9-14] state that the orthogonal dimensions indicate two different aspects: *conceptualization* (columns) and *implementation* (rows). The conceptualization aims to the definition of conceptual models for describing reality, including different abstraction levels. The implementation aims to

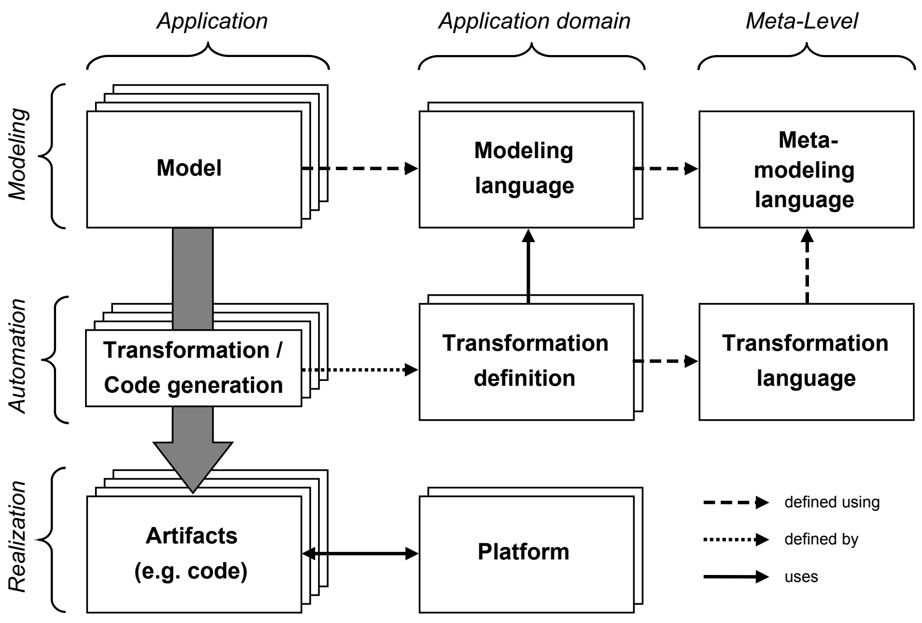


Figure 2.4. Overview of the MDSE methodology [Brambilla et al., 2012, p.10]

the mapping of models to systems; transformation of application models to their running realizations offers several benefits such as reuse of models and platform independence. In this thesis, a generative software development platform which builds on the MDSD paradigm will be considered. The platform, namely *b+m gear*, will be introduced in Section 2.5.

In context of MDSE, *domain-specific languages (DSLs)* can be used for describing issues in specific domains; they are designed especially for such purposes, targeting a certain domain, context, or company [Brambilla et al., 2012, p.13]. A DSL for the Markov4JMeter workload model, as described in Section 2.2, will be introduced in Section 3.1.

2.4.2 Eclipse Modeling Framework

The *Eclipse Modeling Framework (EMF)* is a facility within the *Eclipse* integrated development environment (IDE), for generating code from structured data models [The Eclipse Foundation, 2014a]. Following the MDSD paradigm, it provides a powerful set of tools for creating models and generating corresponding Java classes.

Ecore Meta-Model

The specification of models in EMF is based on a meta-model, namely *Ecore*, which is itself an EMF model [Steinberg et al., 2009]. Figure 2.5 depicts an excerpt of the Ecore meta-model, illustrating essential elements which most closely correspond to the Unified Modeling Language (UML) standard; common base classes have been left away for the sake of clarity [Steinberg et al., 2009]. EMF provides many useful tools for Ecore models by default, targeting miscellaneous standard tasks, such as persistence, serialization, and validation. Ecore models are stored in XML Metadata Interchange (XMI) format [The Eclipse Foundation, 2014a]. Since editing XMI files by hand is an error-prone task, EMF provides several editors which simplify the specification of Ecore models. In particular, editors for displaying models as diagrams, tree structures, or even as syntax-highlighted text are provided.

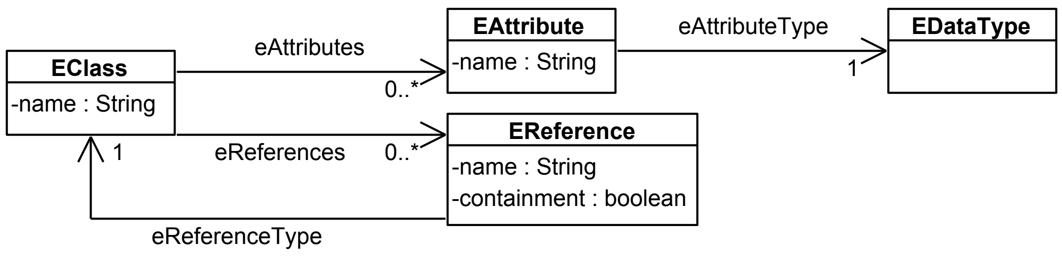
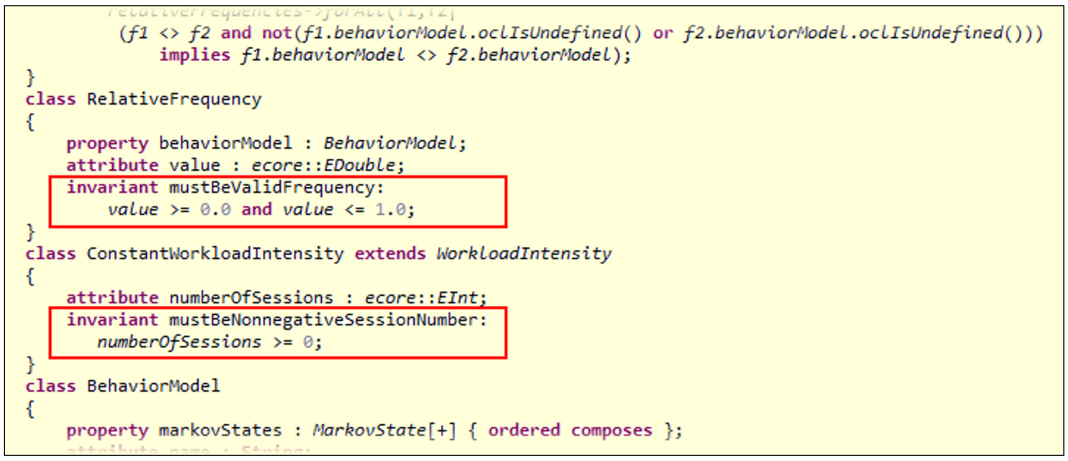


Figure 2.5. A simplified subset of the Ecore meta-model [Steinberg et al., 2009, p.17]



```
        (f1 <> f2 and not(f1.behaviorModel.oclIsUndefined() or f2.behaviorModel.oclIsUndefined()))
            implies f1.behaviorModel <> f2.behaviorModel);
}
class RelativeFrequency
{
    property behaviorModel : BehaviorModel;
    attribute value : ecore::EDouble;
    invariant mustBeValidFrequency:
        value >= 0.0 and value <= 1.0;
}
class ConstantWorkloadIntensity extends WorkloadIntensity
{
    attribute numberOfSessions : ecore::EInt;
    invariant mustBeNonnegativeSessionNumber:
       numberOfSessions >= 0;
}
class BehaviorModel
{
    property markovStates : MarkovState[+] { ordered composes };
```

Figure 2.6. *OCLinEcore Editor* (excerpt) of the EMF, illustrating the definition of OCL constraints.

Modeling OCL Constraints in Ecore

The *Object Constraint Language (OCL)* is a formal language which provides a standard notation for enriching models with additional constraints. Instances of a specific meta-model might still have invalid characteristics regarding to the underlying specification, e.g., improper attribute values which are not conform to certain dependencies between model elements. Hence, the meta-model must be enriched with additional constraints to avoid such undesired instances. Instead of specifying those constraints in a natural language, which would be possibly ambiguous or confusing, expressions can be defined clearly by the use of OCL [Object Management Group, 2012].

OCL constraints for Ecore models are supported by Eclipse. Since the Helios (Eclipse 3.6) release, EMF provides the so-called *OCLinEcore Editor*, which facilitates the input of OCL constraints for Ecore models [The Eclipse Foundation, 2014b]. The text-based editor provides syntax highlighting as well as auto-completion of OCL keywords, simplifying the definition of OCL expressions. It additionally offers the option to assign names to constraints, for related error messages being displayed if violations occur.

Figure 2.6 depicts an excerpt of the OCLinEcore editor. It shows the class definitions for *RelativeFrequency* and *ConstantWorkloadIntensity*, both enrichted with OCL invariants. The marked constraints *mustBeValidFrequency* and *mustBeNonnegativeSessionNumber* ensure that relative frequencies range between 0.0 and 1.0, and that a number of sessions is nonnegative. For this thesis, all constraint names have been selected with a leading *mustBe* prefix for a uniform format.

Parsing Ecore Models with Xtext

Xtext is a framework for the Eclipse IDE which facilitates the implementation of programming languages and DSLs [Bettini, 2013]. The framework can be used for defining a textual DSL with entities corresponding to an underlying Ecore model as described before. Bettini [2013] states that a particular benefit of Xtext is given by the quick implementation process for any DSL, requiring only the definition of a corresponding grammar; the Xtext framework generates related artifacts, such as parser, abstract syntax tree (AST) model, and editor by default. Those artifacts can be customized for individual purposes. In this thesis, Xtext will be used for building a standalone parser which reads data fragments produced in the generation process of an MDSD application. Further details will be discussed in Section 4.1.

2.5 b+m gear Platform

This section describes the b+m gear platform, which constitutes the generative software development platform to be considered in this thesis. The platform is additionally equipped with a generative reference application, namely CarShare, which will be used as a case study system for this thesis. Section 2.5.1 gives an overview of the b+m gear platform, shortly describing its core functionality. Section 2.5.2 discusses architectural aspects of generative b+m gear applications. Section 2.5.3 introduces the CarShare application. Section 2.5.4 outlines the current performance testing methodologies used for b+m gear.

2.5.1 Overview

The MDSD platform *b+m gear* has been developed by b+m Informatik AG [b+m Informatik AG, 2013] and mainly aims to the realization of client-server Web 2.0 applications, mostly based on a service-oriented architecture (SOA); it combines common technical aspects and concepts to avoid individual implementations of frameworks for each of its generative applications [Reimer, 2013, p.4]. Applications generated with b+m gear feature a two- or three-tier architecture, including an additional business workflow layer optionally; those layers will be further discussed in Section 2.5.2.

The software development platform runs in Windows XP or higher and is deployed with an Eclipse IDE providing all required tools. In particular, modeling tools for all architecture layers are included, to follow the MDSD approach described by Stahl and Völter [2005, p.15–17]. The current b+m gear version 3.0.0 has been released in March 2014.

2.5.2 Architecture Layers supported by b+m gear

The architecture supported by b+m gear is designed for the development of highly configurable, customized multi-user systems [Reimer, 2013, p. 4]. By default, it is separated into

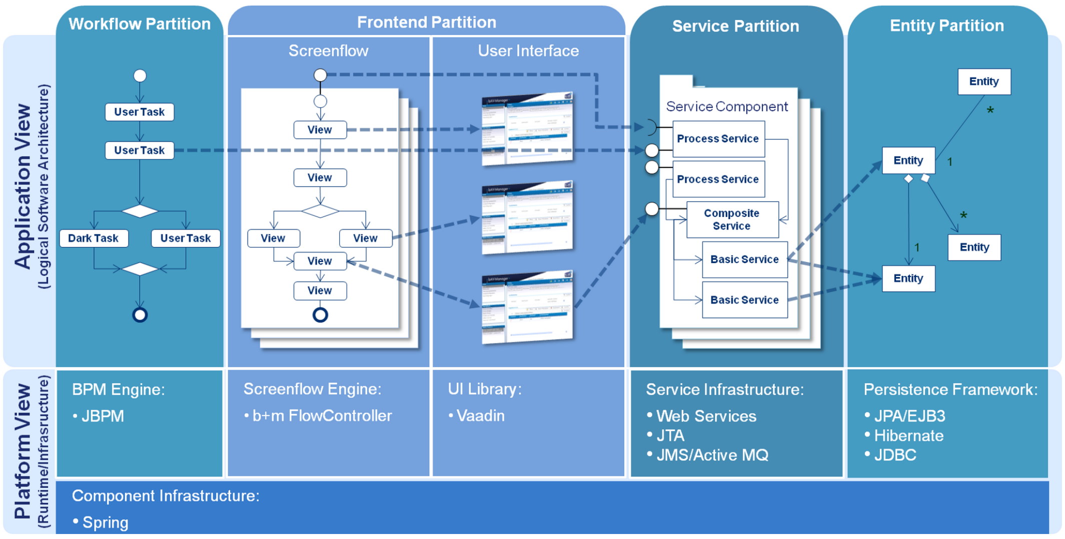


Figure 2.7. b+m gear architecture and runtime [b+m Informatik AG, 2013]

three layers, namely *Frontend (presentation)*, *Service*, and *Entity (persistence)*; additionally, an optional *Workflow* layer is supported, which corresponds to the Business Process Modeling Notation (BPMN) [b+m Informatik AG, 2013]. Reimer [2013, p. 4-5] states that variations of the default architecture are possible, e.g., to achieve direct access between presentation layer and persistence layer. Figure 2.7 gives an overview of the four b+m gear layers, depicting them as *partitions*. For each partition, the underlying runtime technologies are specified in the *Platform View* part of the figure. The *Spring* framework [The Spring Team, 2014] is used as a base container for all partitions, to make the application configuration easily expandable and independent from arbitrary server environments [Reimer, 2013, p. 25]. The partitions are organized as follows:

- ▻ The Entity partition includes the application entities and their relationships. Runtime data is mapped into the database by using a Hibernate [JBoss Community, 2013] implementation which complies to the Java Persistence API (JPA) [Reimer, 2013, p. 25].
- ▻ The Service partition contains the service methods for the business logic. It is important to note that the meaning of the term *service* differs from the meaning within Markov4JMeter: while services in Markov4JMeter can be considered as single use cases as discussed in Section 2.2, services in terms of b+m gear are rather specified from a technical point of view, being understood as interfaces which encapsulate methods for accessing entities via report- and entity-factories. Such methods also serve as transaction controllers for interchanging data via so-called *high value interfaces (HVIs)*, which denote data containers for being used as input/output parameters of service methods [Reimer, 2013, p.25 | 49]. In the remainder of this thesis, services within the meaning of b+m gear

will be shortly referred to as *gear-services*, if the meaning becomes not clear from the context.

The *Service Layer* of the Service partition allows the use of services provided by external remote-systems; it is not part of the b+m gear platform itself [Reimer, 2013, p.25].

- ▻ The Frontend partition constitutes the presentation layer, that is the UI which regulates the interaction between user and application. The Asynchronous JavaScript and XML (AJAX)-based interaction is implemented by using so-called *Views* which denote event-triggered units with underlying *Controllers* for accessing corresponding services. The Views and their underlying Controllers are combined in so-called *Screenflows* which represent reusable units; those units can be invoked hierarchically within the front-end, *synchronous* and *asynchronous* as well [Reimer, 2013, p.25]. In an asynchronous call, control is immediately returned to the invoking instance, allowing the user to perform several tasks at once; in a synchronous call, the invoked unit needs to return an answer until control is given back to the calling instance [Reimer, 2013, p.54–55].
- ▻ The *Workflow* partition allows the definition of application tasks to be performed. Such tasks are specified by their related work flows; in the Frontend partition or services in the Service partition are executed via so-called *BusinessServices* for [Reimer, 2013, p.69].

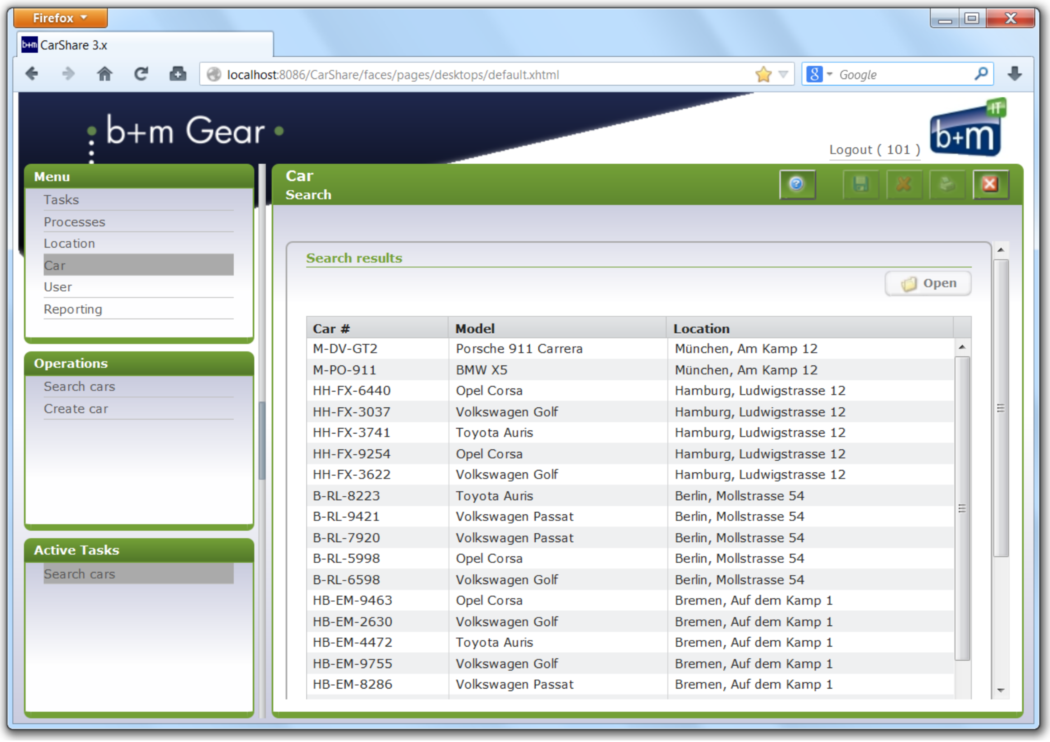
Since version 3.0.0, models in b+m gear are defined by the use of Xtext technologies. Furthermore, the underlying AJAX technology of the front-end changed from JavaServer Faces (JSF) to the *Vaadin* [Vaadin Ltd., 2014] UI library. Testing problems which might arise particularly in context of Vaadin 7 and JMeter will be discussed in Section 2.6.3.

2.5.3 Case Study System CarShare

The generative b+m gear application *CarShare*, developed by b+m Informatik AG, is considered to be used as a case study system for this thesis. The workflow-based system is an example for applications being generated with b+m gear, and it serves as a reference application with an appropriate configuration of the model partitions. Figure 2.8 shows a screenshot of CarShare, depicting a *Search* Screenflow which includes a *Search results* View for registered cars.

2.5.4 Current Performance Testing Methodology for b+m gear

Performance tests which are conducted by b+m Informatik AG on b+m gear applications currently depend on hand-crafted test scripts for the testing tool JMeter. These scripts need to be adjusted for any changes made in the targeted applications; in particular, script updates might become necessary, whenever underlying technology changes, since protocol information is also changed in such cases generally. Hence, the generation of appropriate scripts, to avoid their time-consuming manual construction, is one of the goals of this thesis, as indicated in Section 1.2.



Car #	Model	Location
M-DV-GT2	Porsche 911 Carrera	München, Am Kamp 12
M-PO-911	BMW X5	München, Am Kamp 12
HH-FX-6440	Opel Corsa	Hamburg, Ludwigstrasse 12
HH-FX-3037	Volkswagen Golf	Hamburg, Ludwigstrasse 12
HH-FX-3741	Toyota Auris	Hamburg, Ludwigstrasse 12
HH-FX-9254	Opel Corsa	Hamburg, Ludwigstrasse 12
HH-FX-3622	Volkswagen Golf	Hamburg, Ludwigstrasse 12
B-RL-8223	Toyota Auris	Berlin, Mollstrasse 54
B-RL-9421	Volkswagen Passat	Berlin, Mollstrasse 54
B-RL-7920	Volkswagen Passat	Berlin, Mollstrasse 54
B-RL-5998	Opel Corsa	Berlin, Mollstrasse 54
B-RL-6598	Volkswagen Golf	Berlin, Mollstrasse 54
HB-EM-9463	Opel Corsa	Bremen, Auf dem Kamp 1
HB-EM-2630	Volkswagen Golf	Bremen, Auf dem Kamp 1
HB-EM-4472	Toyota Auris	Bremen, Auf dem Kamp 1
HB-EM-9755	Volkswagen Golf	Bremen, Auf dem Kamp 1
HB-EM-8286	Volkswagen Passat	Bremen, Auf dem Kamp 1

Figure 2.8. Screenshot of the CarShare application

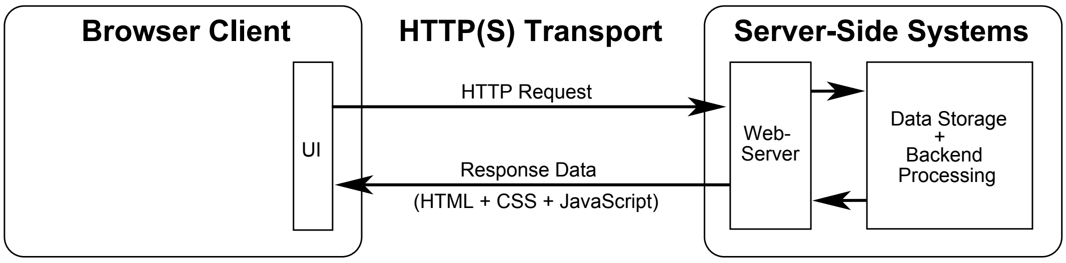
2.6 Challenges in Testing AJAX Applications with JMeter

This section illustrates the challenges that have been identified in terms of testing AJAX applications with JMeter. It states out the conditions which must hold for the construction of appropriate Test Plans and discusses whether these conditions hold in connection with applications building on the AJAX-based Vaadin framework. Finally, an alternative testing methodology will be discussed. Section 2.6.1 outlines the principal differences between the classic Web application model and the AJAX Web application model. Section 2.6.2 describes the capabilities of JMeter for testing AJAX applications. Section 2.6.3 presents several issues which have been identified in context of conducting performance tests on Vaadin-based applications. Section 2.6.4 outlines an alternative testing methodology which aims to the circumvention of Frontend-related issues.

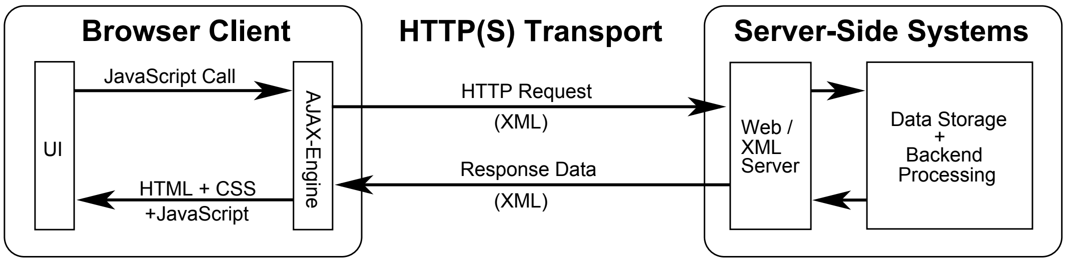
2.6.1 Classic Web Application Model vs. AJAX Web Application Model

Classic Web applications follow a simple approach of client-server communication, which requires a lot of traffic and response time. For increasing the performance as well as the usability of Web systems, AJAX systems build particularly on JavaScript technology, allowing the user to interact with the application more efficiently.

Figure 2.9 illustrates the difference between the classic Web application model and the AJAX Web application model. In the classic model, an HTTP request triggered by the user is sent straight from the browser client UI to the server. The server receives that request and sends corresponding response data back to the client. A significant disadvantage of this approach is the requirement to reload a complete Web page for every piece of data, limiting interactivity and affecting the user experience in a negative manner [Paulson, 2005]. Furthermore, the amount of response data, which includes HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and/or JavaScript fragments, requires a lot of traffic. In the AJAX Web application model, the user triggers via browser UI a JavaScript call, which is already processed on client side by the so-called *AJAX engine*. Such engine, which is itself written in JavaScript, is loaded initially whenever a Web page is requested; Web page content is displayed by that engine, instead of being loaded



(a) Classic Web application model



(b) AJAX Web application model

Figure 2.9. Classic and AJAX Web application models, based on Paulson [2005]; Yang et al. [2007]

completely for each request [Paulson, 2005]. In particular, the engine forwards a request as an *XMLHttpRequest* to the server, decreasing the required traffic volume for the purpose of faster responses; it even supports *asynchronous* calls to be sent to the server, allowing the user to interact continuously with the application while data is updated [Paulson, 2005]. Response data is processed accordingly, when being received by the AJAX engine: it is transformed to HTML/CSS/JavaScript content for being displayed in the client UI.

The interaction between UI and AJAX engine on client side limits the realization of performance tests using JMeter, since appropriate Test Plans can be constructed only under certain conditions. This will be discussed in the following section.

2.6.2 JMeter Capabilities for AJAX Applications

JMeter supports performance tests targeting systems which follow the classic Web application model. In contrast, the capabilities of JMeter in terms of AJAX technology are very limited. Generally, the configuration of Test Plans targeting AJAX applications is difficult, since JMeter does not process JavaScript which is embedded in HTML pages [The Apache Software Foundation, 2014b]. In particular, direct calls of JavaScript functions in an AJAX engine are not provided; instead, it is recommended [The Apache Software Foundation, 2014b] to use the JMeter proxy server facility for creating the Samplers which are required for sending appropriate requests. Figure 2.10 gives an overview of the current situation, depicting the JMeter application including two alternatives for building Test Plans, both separated by a dotted line: using an editor for specifying Test Plans manually, or using a proxy server for recording requests to be replayed. Through the fact that JavaScript calls are not provided by JMeter, the manual specification of Test Plans targeting functions of an AJAX engine is not possible. Alternatively, the JMeter proxy server facility might be used for recording requests intercepted from a client/server communication. However, Figure 2.10 even shows the drawback of the latter approach: no AJAX engine is considered when recorded requests are replayed from a Test Plan. In particular, those requests are replayed "as they are", assuming that values of each Sampler are valid for arbitrary user sessions. If the AJAX engine computes session-related information at runtime, that information is not updated in Test Plan Samplers respectively their associated requests. Consequently, invalid information is transmitted to the server in such cases generally. Any modification of recorded Test Plan Samplers might even lead to invalid testing results for the same reasons.

In summary, JMeter currently can only be used for testing AJAX systems by utilizing the proxy server facility for intercepting default values and modifying the associated Samplers afterward; an additional condition which must hold thereby is that relevant information, such as client-server synchronization data, of each request does not depend on values which are computed during runtime. Challenges concerning this condition have been experienced with the Vaadin 7 framework; this will be discussed in the following section.

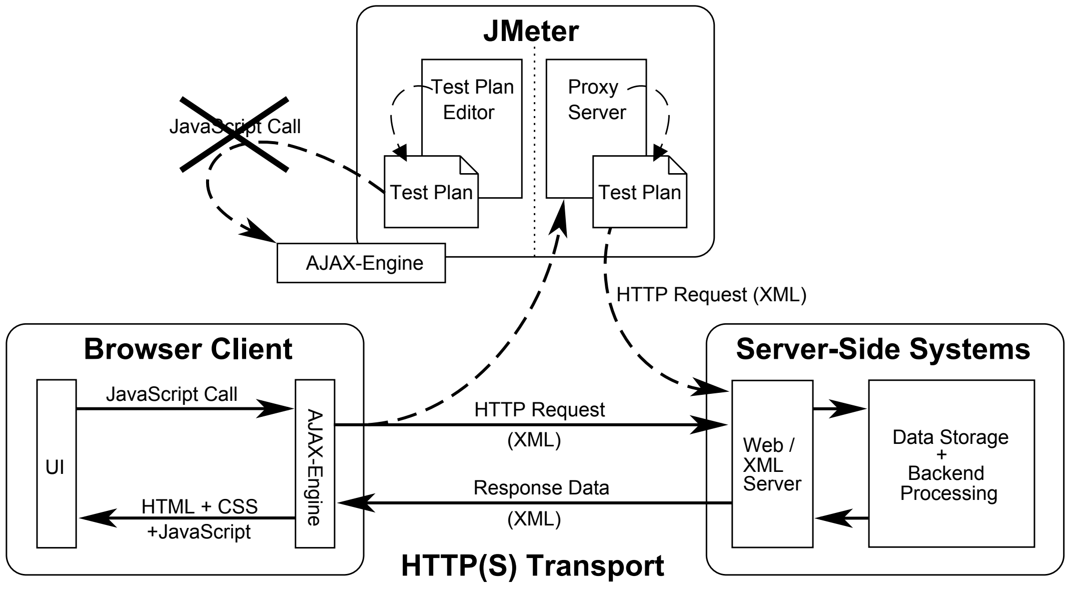


Figure 2.10. JMeter capabilities for AJAX Web applications, based on Figure 2.9b

2.6.3 Vaadin/JMeter Testing Issues

Vaadin is a Java-based framework for developing AJAX Web applications, particularly aiming to large enterprise systems with focus on ease of use, re-usability, and extensibility [Grönroos, 2014]. On grounds of efficiency, the framework uses *JavaScript Object Notation (JSON)*, which is a compact data interchange format for the communication between server and clients; prior to Vaadin 5, XML has been used instead [Grönroos, 2012, p.32]. Figure 2.11 shows some example data of JSON-formatted Vaadin 7 requests which have been intercepted with a proxy server. These requests have been triggered through events fired by the use of certain UI widgets, such as buttons.

Vaadin builds on client-server synchronization, that is, an update on one tier must be reflected on the other [Fränkel, 2013, p.60]. For this purpose, the framework uses *synchronization keys* which are generated by the server and passed to the regarding clients on each request; a key is sent back to the server within the next triggered client-request; if not, the current session is restarted by the server [Fränkel, 2013, p.60]. In particular, so-called *Connector IDs* are used for the communication between client widgets and components of the server application. The (simplified) AJAX Web application model for Vaadin is illustrated in Figure 2.12: each HTTP request transmits the Connector ID for synchronization purposes from client to server; if the ID does not match, the synchronization fails on server side.

Connector IDs denote integer values, sequentially generated by the server. They are depicted by example in Figure 2.11 as the leading integer numbers of each request data

```
init [["42","com.vaadin.shared.ui.button.ButtonServerRpc","click",["altKey":false,
"type":"1", "relativeX":"20", "relativeY":"8", "shiftKey":false, "metaKey":false,
"ctrlKey":false, "button":"LEFT", "clientY":"102", "clientX":"49"]]]

                ________________

init [["43","com.vaadin.shared.ui.button.ButtonServerRpc","click",["altKey":false,
"type":"1", "relativeX":"50", "relativeY":"11", "shiftKey":false, "metaKey":false,
"ctrlKey":false, "button":"LEFT", "clientY":"126", "clientX":"79"]]]

                ________________

init [["155","v","v",["filter",["s",""]]],["155","v","v",["page",["i","-1"]]]]
```

Figure 2.11. Examples of JSON-formatted requests for a Vaadin 7 application

set, i. e., 42, 43, and 155 respectively. For debugging purposes, the Vaadin UI library offers to set these IDs explicitly via *setId()* method. However, this does not work consistently anymore since release 7 of Vaadin. In contrast to earlier Vaadin versions, Connector IDs defined that way are ignored, and only server-generated Connector IDs are used instead. Consequently, the client-server synchronization is not controllable for the tester anymore. This is a substantial barrier in terms of JMeter Test Plan generation, since Connector IDs are generated dynamically by the server and unavailable until runtime. In particular, these IDs are even unavailable in the generation process of any application, so that dedicated Test Plans cannot be equipped with this information, since the condition for relevant information as discussed in Section 2.6.1 does not hold. A manual inspection of code, to determine what the JavaScript is doing on client side, as recommended by The Apache Software Foundation [2014b], is also difficult through the fact that the AJAX engine uses obfuscation. Even though an ID assignment scheme might be possibly determined that way, its emulation remains to be difficult, if ID assignments depend on the UI structure including widget interdependencies which had to be emulated, too. Figure 2.13 shows an

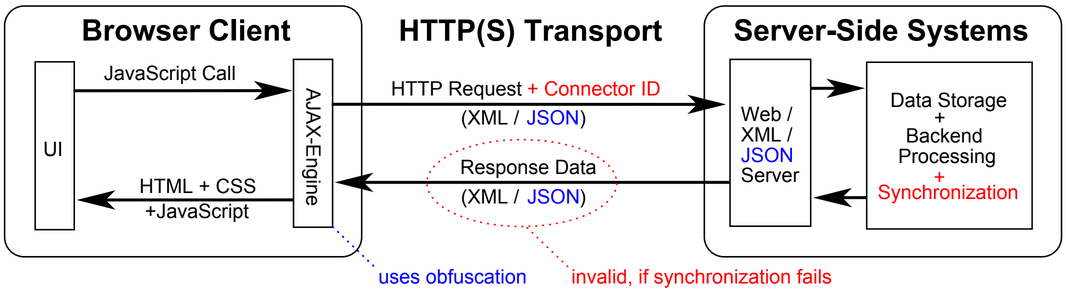


Figure 2.12. (Simplified) AJAX Web application model for Vaadin, based on Figure 2.9b

```
24.01.2014 16:18:49 com.vaadin.server.communication.ServerRpcHandler
parseInvocation
WARNING: RPC call to v.v received for connector 124 but no such
connector could be found.  Resynchronizing client.  24.01.2014 16:18:49
com.vaadin.server.communication.ServerRpcHandler parseInvocation
WARNING: RPC call to v.v received for connector 124 but no such
connector could be found.  Resynchronizing client.  24.01.2014 16:18:49
com.vaadin.server.communication.ServerRpcHandler parseInvocation
WARNING: RPC call to com.vaadin.shared.ui.button.ButtonServerRpc.click received
for connector 127 but no such connector could be found.  Resynchronizing client.
```

Figure 2.13. Server output indicating synchronization errors in a Vaadin 7 application

error output on server side, indicating synchronization errors in a Vaadin 7 application; the response data of such requests is generally invalid.

Another aspect to be considered is that the Vaadin UI uses background tasks which might substantially influence the performance testing results. For example, if a user requests some content data of a multi-page table widget on client side, the processing Vaadin application on server side does not only return a single chunk of data for the currently selected page; instead, it generally runs additional background tasks for performance optimization, such as caching operations, for returning requested content data faster for each table page. If performance of service functions shall be determined, it might remain unclear whether certain measurement results are caused by the functions themselves or by the overhead of the Vaadin framework.

2.6.4 Conclusions for b+m gear Performance Tests

The Vaadin/JMeter issues discussed in Section 2.6.3 constitute general challenges in testing AJAX applications with JMeter. However, they raise the question whether it would not be more useful to circumvent the UI in tests which build on user behavior emulation for bottleneck or error identification purposes; weaknesses in the (self-written) underlying service-related code could be located more precisely that way. This approach will be followed in this work, owing the fact that a solution for the previously discussed AJAX challenges is out of scope for this thesis; that is, the framework to be created applies Screenflow-based Java requests locally on gear-service methods, instead of sending HTTP requests to a remote server. Aspects like Remote Procedure Call (RPC) will be left out in the first attempt. Hence, the following section introduces the capabilities of JMeter in terms of invoking Java methods locally.

2.7 Java Method Invocations via JMeter

Invoking Java methods from a JMeter Test Plan can be implemented in various ways, depending on the Sampler types to be used. JMeter offers three essential Sampler types which support the direct execution of Java code. Appropriate Samplers are of type *Beanshell Sampler*, *Java Request Sampler*, and *JUnit* Sampler. These Samplers will be introduced in this section, including a discussion of their individual benefits and drawbacks, which is based on [newspaint, 2014]. The section closes with a short comparison overview.

2.7.1 BeanShell Sampler

The name of this Sampler results from the scripting language *BeanShell* which is defined by its author as follows [Niemeyer, 2014]:

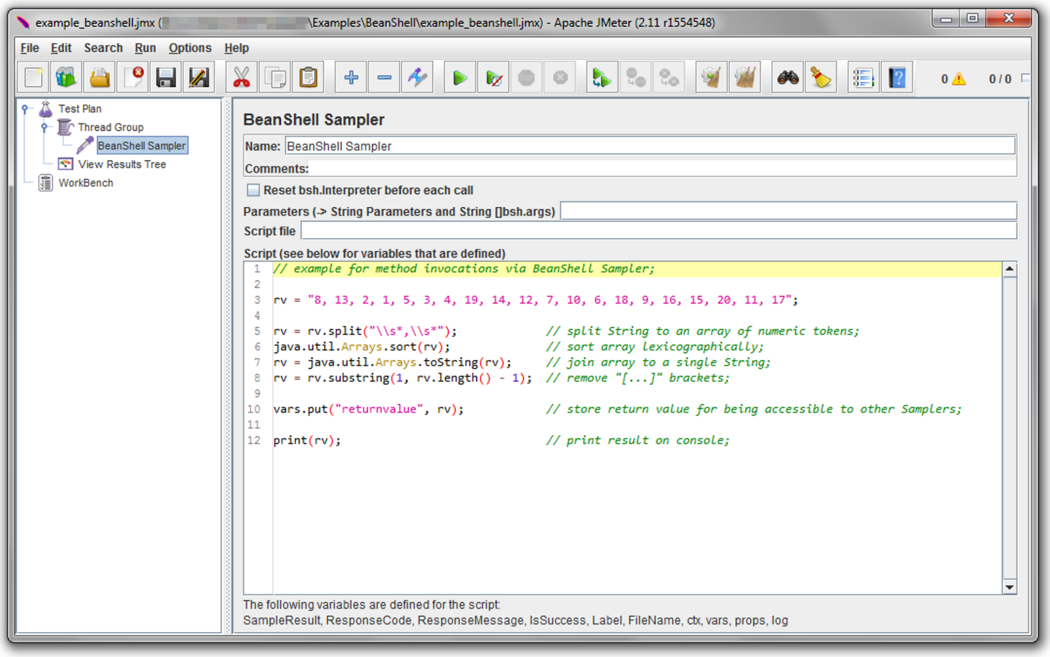
> *"BeanShell is a small, free, embeddable Java source interpreter with object scripting language features, written in Java. BeanShell executes standard Java statements and expressions, but also extends Java into the scripting domain with common scripting language conventions and syntax. BeanShell is a natural scripting language for Java."*

A JMeter *BeanShell Sampler* offers a simple and flexible way to define Java method invocations, logic control statements, and expressions to be evaluated while test runtime. Figure 2.14 depicts a screenshot of such a Sampler, including several script lines by example. As an additional feature, the import of predefined script files is provided by the BeanShell Sampler as well as syntax highlighting. An important benefit is the full access to Test Plan variables and the option to define new variables as well. This is a core requirement for sharing return values between Samplers and implementing persistence of certain information within a JMeter test run. A single BeanShell Sampler is associated with one test only, but it offers flexibility through scripting techniques. On the other hand, this technique is probably the biggest drawback of this Sampler type regarding performance tests; the scripting engine constitutes a relative large overhead, affecting fine-grained time measurements of local tests in particular.

Table 2.1 shortly summarizes the discussed issues. The BeanShell Sampler is a highly flexible element with simple usability, but its main drawback is the low performance.

Table 2.1. Benefits and drawbacks of a JMeter BeanShell Sampler

BeanShell Sampler
+ simple usability (definition of tasks via scripting language). + full access to Test Plan variables. + high flexibility. - low performance through scripting engine. - includes only one test in a single Sampler.


example_beanshell.jmx ([redacted]\Examples\BeanShell\example_beanshell.jmx) - Apache JMeter (2.11 r1554548)
File Edit Search Run Options Help
0
0 / 0
Test Plan
Thread Group
BeanShell Sampler
View Results Tree
WorkBench

BeanShell Sampler

Name: BeanShell Sampler

Comments:

Reset bsh.Interpreter before each call

Parameters (-> String Parameters and String []bsh.args)

Script file

Script (see below for variables that are defined)

```
// example for method invocations via BeanShell Sampler;

rv = "8, 13, 2, 1, 5, 3, 4, 19, 14, 12, 7, 10, 6, 18, 9, 16, 15, 20, 11, 17";

rv = rv.split("\\s*,\\s*");            // split String to an array of numeric tokens;
java.util.Arrays.sort(rv);             // sort array lexicographically;
rv = java.util.Arrays.toString(rv);    // join array to a single String;
rv = rv.substring(1, rv.length() - 1); // remove "[...]" brackets;

vars.put("returnvalue", rv);           // store return value for being accessible to other Samplers;

print(rv);                             // print result on console;
```

The following variables are defined for the script:
SampleResult, ResponseCode, ResponseMessage, IsSuccess, Label, FileName, ctx, vars, props, log

Figure 2.14. Screenshot of a configuration panel for a BeanShell Sampler in JMeter, including several script lines by example.

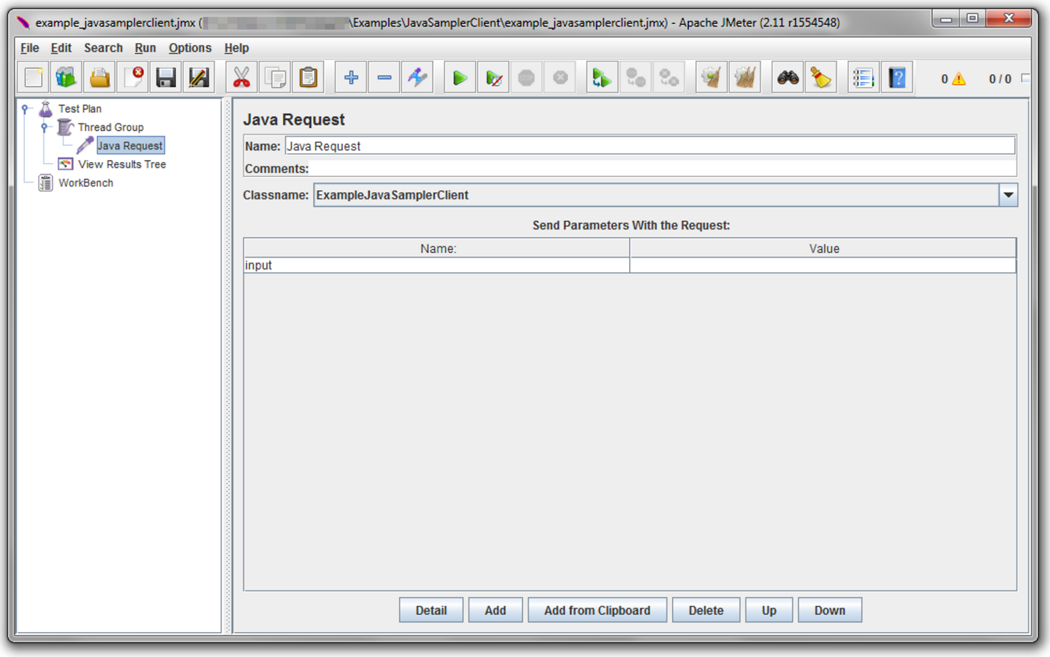
2.7.2 Java Request Sampler

As indicated by its name, the idea of a *Java Request Sampler* is to control a client class via Test Plan by telling that class which Java method is to be executed next [The Apache Software Foundation, 2014a]. This implies a high performance, since the effort for controlling the client class is relatively low, and method invocations can be done by the Java Virtual Machine (JVM) without any Sampler-specific scripts being parsed before.

A class which is intended to receive a Sampler request for starting related test code is shortly referred to as *Java Sampler Client*. For being identified by JMeter as such, it must either implement the interface `JavaSamplerClient.java` or -even simplier- extend the abstract class `AbstractJavaSamplerClient.java`, both provided by the JMeter API. Listing 2.1 shows an example of a Java Sampler Client. Method `runTest()` serves as entry point, and method `getDefaultParameters()` provides the supported client parameters. The result sample to be returned by a Java Sampler Client contains a response code and response data as well. The response code indicates whether a test was successful, and the (optional) response data denotes a `String` representation of the object which might have been computed within a test. A Java Sampler Client has to be put into a Java Archive (JAR) after being compiled, and the JAR has to be moved to the `lib/ext/` folder of the JMeter installation directory for

```java
public class ExampleJavaSamplerClient extends AbstractJavaSamplerClient {

    @Override
    public SampleResult runTest (JavaSamplerContext javaSamplerContext) {

        // results report to be returned;
        SampleResult result = new SampleResult();

        // record the start time of sample;
        result.sampleStart();

        try {
            // invoke the test to be done by the Sampler;
            String responseData = ...

            result.setDataType(SampleResult.TEXT);
            result.setResponseData(responseData, "UTF-8");
            result.setResponseMessage("Sampler succeeded.");
            result.setResponseOK();
            result.setResponseCodeOK();

        } catch (Exception ex) {
            result.setResponseMessage("Sampler failed: " + ex.getMessage());
            result.setResponseCode("500");

        } finally {
            // record the end time of sample, calculate elapsed time;
            result.sampleEnd();
        }

        return result;
    }

    @Override
    public Arguments getDefaultParameters () {

        // define parameter(s) accepted by this type of JavaSamplerClient;
        Arguments defaultParameters = new Arguments();
        defaultParameters.addArgument("input", "");

        return defaultParameters;
    }
}
```

Listing 2.1. Example for a Java Sampler Client, based on [newspaint, 2014]


example_javasamplerclient.jmx ([redacted]\Examples\JavaSamplerClient\example_javasamplerclient.jmx) - Apache JMeter (2.11 r1554548)
File Edit Search Run Options Help
0 ⚠ 0 / 0
Test Plan
Thread Group
Java Request
View Results Tree
WorkBench
Java Request
Detail
Add
Add from Clipboard
Delete
Up
Down

Name: Java Request

Comments:

Classname: ExampleJavaSamplerClient

Send Parameters With the Request:

Name:	Value
input	

Figure 2.15. Screenshot of a configuration panel for a Java Request Sampler in JMeter, illustrating several options for an *Example Java Sampler Client*.

being recognized by the tool. The same procedure must be applied to the targeted test application. The implementation process of a Java Sampler Client is a drawback regarding to the usability of Java Request Samplers, since code has to be compiled, packed and installed whenever any changes have been made.

Test Plan variables are fully accessible by Java Sampler Clients. Each thread is associated with a context object from which its “local” variables can be requested, and new variables can be defined as well. Like a BeanShell Sampler, a single Java Request Sampler is associated with one test only. In particular, its regarding set of parameters is appropriate for that one test only. For any other testing purposes, alternative Java Sampler Clients with appropriate parameters must be implemented. However, through the fact that almost any Java code is allowed to be implemented in a Java Sampler Client, a Java Request Sampler is very flexible. For example, a dedicated Sampler parameter might indicate the testing method to be invoked next.

Figure 2.15 shows a screenshot of a Java Request Sampler in JMeter, illustrating configuration options for the *Example Java Sampler Client* which has introduced before. Note that the *input* parameter is available since it is returned by the `getDefaultParameters()` method as shown in Listing 2.1. All provided parameters must be predefined that way,

Table 2.2. Benefits and drawbacks of a JMeter Java Request Sampler

Java Request Sampler
+ high performance.
+ full access to Test Plan variables.
+ high flexibility.
- complex usability (implementation/compilation of client required).
- includes only one test in a single Sampler.

JMeter does not allow to declare additional parameters while runtime, although the button panel at the bottom of the configuration panel indicates this. JMeter provides two standard Java Samplers, namely *JavaTest* and *SleepTest*, both located in the JMeter API package `org.apache.jmeter.protocol.java.test`. They provide helpful information by their source codes, regarding to the usage of JMeter Java Samplers.

Table 2.2 shortly summarizes the discussed issues. The Java Request Sampler is a highly flexible element with good performance, but its main drawback is the complex usability.

2.7.3 JUnit Sampler

The *JUnit* framework is a simple, programmer-oriented instance of the xUnit architecture which is intended for writing repeatable tests [JUnit Team, 2014]. A *JUnit Sampler* supports the integration of such tests into a JMeter Test Plan. Any Java class which represents a JUnit test case, as illustrated by example in Listing 2.2, might be used. In particular, no dedicated JMeter interface or abstract class must be additionally implemented or extended, respectively. A Testing method defined in such a class can be individually selected as an invocation candidate in the JUnit Sampler configuration panel of the JMeter GUI. Each Sampler invokes exactly one testing method. Figure 2.16 depicts a screenshot of the configuration panel for a JUnit Sampler, illustrating several configuration options. The low effort for invoking JUnit test cases implies a high performance of this Sampler type. Multiple tests might be included in a single class by implementing related methods. These methods indicate multiple tests being contained in a single JUnit Sampler, to be selected regarding to the individual purpose. However, the implementation of test cases has to

Table 2.3. Benefits and drawbacks of a JMeter JUnit Sampler

JUnit Sampler
+ high performance.
+ might include multiple tests in a single Sampler.
- complex usability (implementation/compilation of test case required).
- no access to Test Plan variables.
- low flexibility.

```java
import junit.framework.TestCase;

public class JUnitExample extends TestCase {

    private int i = 4;

    // name-prefix "test" is required for JUnit testing method identification;
    public void testNumberIsOdd () {

        assertTrue("i is an odd number", i % 2 == 1);
    }

    public void testNumberIsEven () {

        assertTrue("i is an even number", i % 2 == 0);
    }
}
```

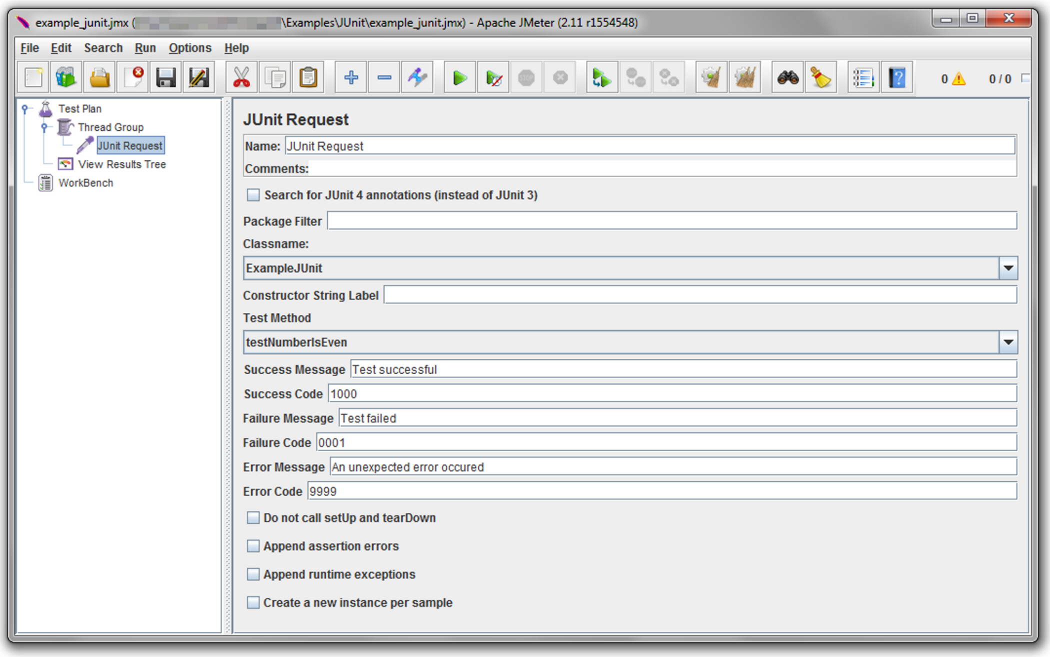
Listing 2.2. Example for a JUnit Test Case

be done following a similar process as required for Java Request Samplers. Code has to be implemented, compiled, packed, and moved as JAR to the `/ext/junit` folder of the JMeter installation directory for being recognized by the tool. The same procedure must be applied to the targeted test application. This is a drawback regarding to the usability of JUnit Samplers. Missing access to Test Plan variables as well as missing parameters for testing functions decrease the flexibility of JUnit Samplers.

Table 2.3 shortly summarizes the discussed issues. The JUnit Sampler has high performance, but its main drawback is the very low flexibility.

2.7.4 Comparison of Samplers and Conclusion

Table 2.4 gives a comparison overview for the introduced Samplers. All samplers additionally provide invocation of private methods. Due to its low flexibility, the JUnit Sampler is unsuitable for being used in terms of Test Plan generation, since it particularly does not provide parameters being passed to testing functions. The BeanShell Sampler is easy to use, and it provides high flexibility, which qualifies it to be used for Test Plan generation. However, to circumvent the performance overhead of the scripting engine, the Java Request Sampler constitutes a good alternative in combination with Java Reflection; further details will be discussed in Section 4.2.


example_junit.jmx ([redacted]\Examples\JUnit\example_junit.jmx) - Apache JMeter (2.11 r1554548)

File Edit Search Run Options Help

Test Plan
Thread Group
JUnit Request
View Results Tree
WorkBench

JUnit Request

Name: JUnit Request

Comments:

Search for JUnit 4 annotations (instead of JUnit 3)

Package Filter

Classname:

ExampleJUnit

Constructor String Label

Test Method

testNumberIsEven

Success Message: Test successful

Success Code: 1000

Failure Message: Test failed

Failure Code: 0001

Error Message: An unexpected error occured

Error Code: 9999

Do not call setUp and tearDown

Append assertion errors

Append runtime exceptions

Create a new instance per sample

Figure 2.16. Screenshot of a configuration panel for a JUnit Sampler in JMeter, illustrating several test configuration options.

Table 2.4. Comparison of JMeter Samplers

	BeanShell	**Java Request**	**JUnit**
Usability	simple	complex	complex
Performance	low	high	high
Max. # of tests in one Sampler	1	1	∞
Access to Test Plan variables	full	full	–
Flexibility	high	high	low

2.8 Performance Testing Approach in DynaMod

This section illustrates the performance testing approach which has been evolved in context of the research-project *DynaMod* [van Hoorn et al., 2011; Schulz, 2013; Schulz et al., 2014]. Its description is intended to make clear how much effort it requires in general to construct model-based tests manually, and it aims to the need of test generation which is one of the main goals of this thesis. Moreover, the approach is a source for the idea of this thesis. It uses the Markov4JMeter workload generation model as introduced in Section 2.2, and it associates Markov States with use cases taken from the behavior specification of the targeted SUT.

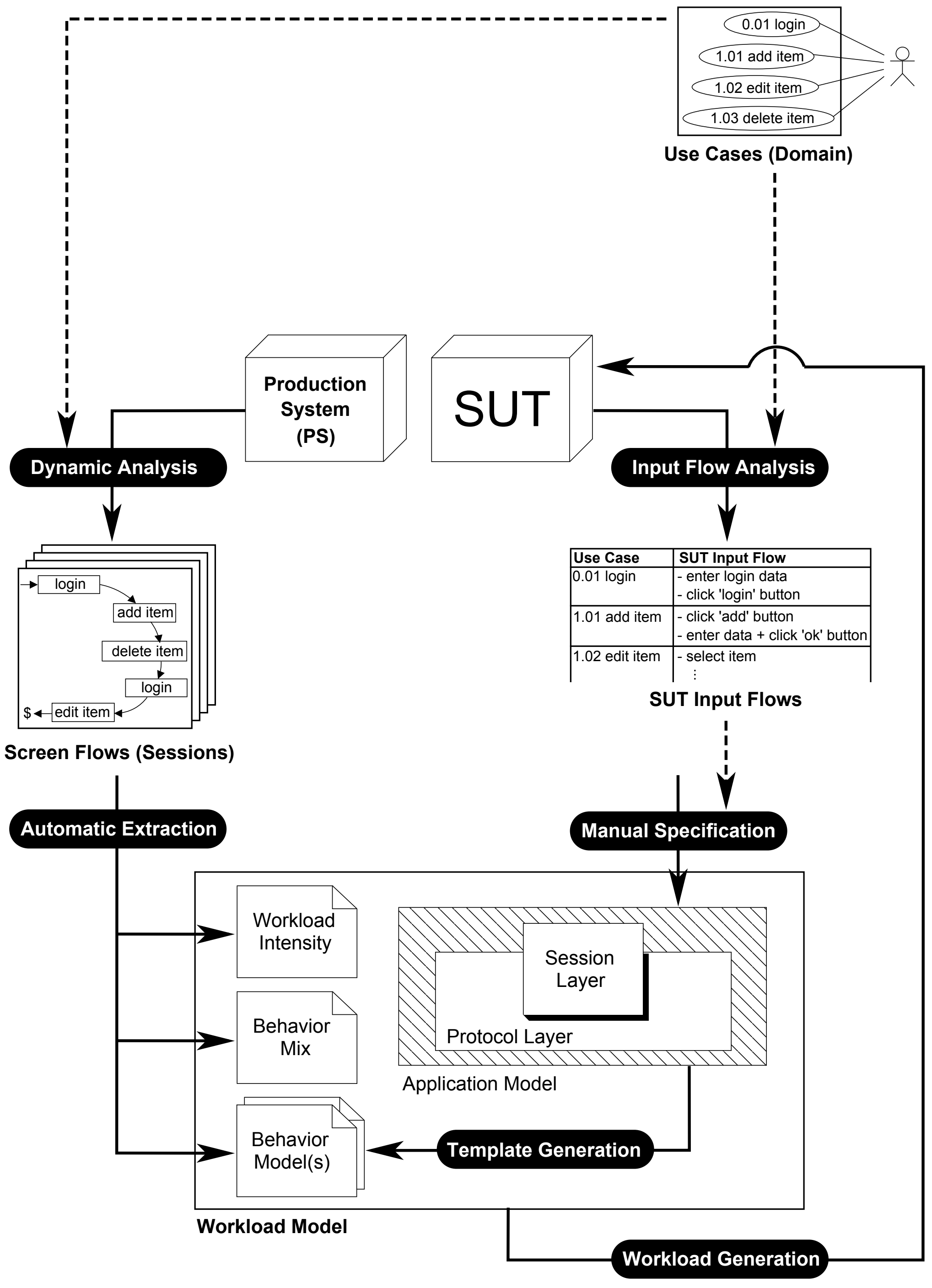
Figure 2.17 gives an overview of the testing approach, depicting the included artifacts and relations between them. Dashed arrows denote dependencies, and solid arrows denote transitions between artifacts, each labeled with a descriptive name of a related task to be performed for taking the transition. The production system (PS) and the SUT are given systems, whereas the PS denotes a system which has been already in productive operation, while the SUT does not necessarily need to be so. For example, the SUT might be a modernized prototype of the PS, to be tested before it goes into field deployment. PS and SUT even might denote the same system, that is a production system whose performance shall be tested. However, both systems are depicted separately for illustration purposes. The approach includes the following six tasks [Schulz, 2013; Schulz et al., 2014]:

- *Analysis of input flow.* As stated before, Markov States are associated with use cases. Each use case indicates a certain part of functionality provided by the SUT. For specifying the requests to be sent in a Markov State of a Test Plan, the input activities performed by a (virtual) user for executing the related use case need to be analyzed and determined. This task results in a mapping between use cases and their related activity sequences, which are shortly referred to as *input flows*. It generally takes much effort to create such a mapping, due to complexity depending on the number of use cases and their variations. However, the determination of input flows is fundamental for the manual specification of the Application Model.

- *Manual specification of an Application Model.* As described in Section 2.2, an Application Model consists of a Session Layer and a Protocol Layer which define valid use case execution sequences and protocol details respectively. The EFSM of the Session Layer can be constructed by analyzing the validity of use case execution sequences, which can be derived from the behavior specification of the considered system. The Protocol Layer can be constructed on the basis of the SUT input flows table. Roughly spoken, an Application Model corresponds to the core structure of a Markov4JMeter Test Plan, which includes all of this information as well. Hence, the task of specifying the Application Model manually can be associated with constructing a corresponding Markov4JMeter Test Plan structure by hand.

- *Generation of a template for Behavior Models.* In Section 2.3.2, the Markov Session Controller of the Markov4JMeter add-on, including its option for generating a template for Behavior Models, has been already introduced. The controller's provided option allows the generation of an empty transition probability matrix over the Markov States which have been specified in the regarding Test Plan. Such a matrix template is stored as a CSV formatted file. This task of the approach simply aims to the use of the related option in JMeter/Markov4JMeter for creating a matrix template, to be filled with actual probability values in the following tasks.

- *Dynamic analysis of the PS.* This task includes monitoring activities for obtaining so-called *screen flows*, as indicated in Figure 2.17. Screen flows describe user traces, that is activity



Use Case	SUT Input Flow
0.01 login	- enter login data - click 'login' button
1.01 add item	- click 'add' button - enter data + click 'ok' button
1.02 edit item	- select item ⋮

Figure 2.17. Overview of the load testing approach in DynaMod, based on Schulz [2013]

sequences of users depending on triggered events, e.g., button clicks or input of data. By setting certain monitoring points in the PS, those user traces can be retrieved and utilized as indicators for behavioral patterns. Values which describe these patterns, e.g., transition probabilities and think times, can be extracted from these traces, serving as input parameters for the workload model.

▻ *Automatic extraction of workload model parameters*. The input parameters for the workload generation model can be retrieved from the screen flows which have been gained through a dynamic analysis of the PS. In particular, the Behavior Models including transition probabilities and think times, the Behavior Mix, and workload intensity can be retrieved from these traces. A dedicated tool, namely *Behavior Model Extraction Tool*, has been developed in context of the DynaMod project. The tool reads a template CSV file for Behavior Models and fills it with probability values extracted from a given set of screen flows.

▻ *Workload generation*. In this task, the workload targeting the SUT is generated, based on the given model. It is assumed that the workload generation itself is done via JMeter, using the Markov4JMeter add-on as introduced in Section 2.3.2.

It is important to note that the analysis of input flows and the specification of the workload model both require a huge amount of manual work, to be preferably substituted through a generative process. This has been already discussed in terms of the goals of this work. The envisioned approach which will be described in the next chapter, mainly aims to this issue.

Chapter 3

M4J-DSL and JMeter Test Plan Creation

This chapter discusses the *JMeter Test Plan Creation* part of the implementation approach, as illustrated in the lower right corner of Figure 1.1. An explicit overview of the JMeter Test Plan creation process, illustrating the assembly status of the included parts, is given in Figure 3.1. Green check marks denote artifacts respectively transformation input/output pipes which have been successfully implemented. Yellow exclamation marks indicate open issues which can be solved with additional effort. The marked parts will be covered in the following discussion. Section 3.1 introduces the Markov4JMeter Domain-Specific Language (M4J-DSL) which represents a meta-model for Markov4JMeter workload models. Section 3.2 presents a framework for transforming M4J-DSL models to JMeter Test Plans.

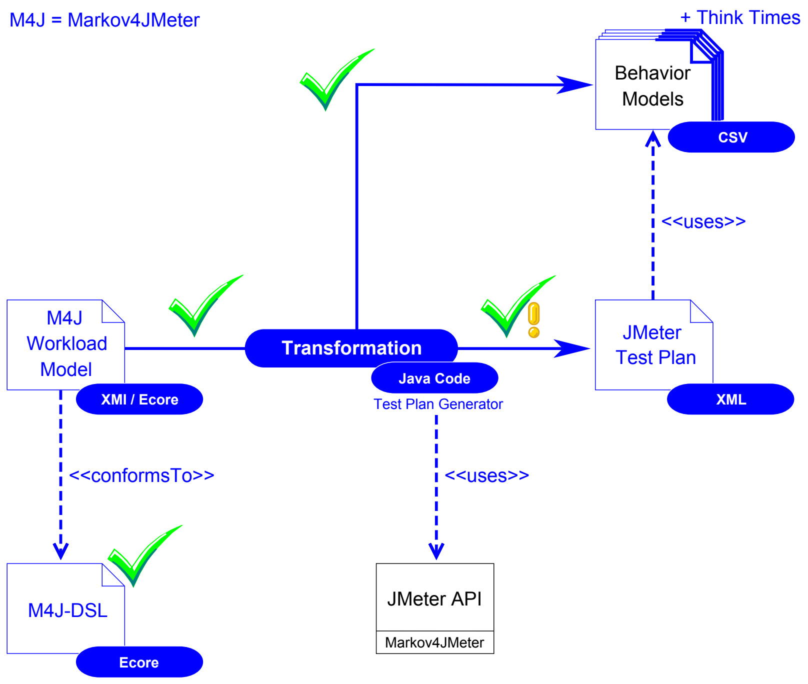


Figure 3.1. Overview of the JMeter Test Plan creation process, illustrating the assembly status of the included parts.

3.1 Markov4JMeter Domain-Specific Language

The specification of the *Markov4JMeter Domain-Specific Language (M4J-DSL)* is based on a meta-model which describes the structure of Markov4JMeter workload models, as introduced in Section 2.2. It defines the entities and their relationships in particular. Constraints formulated in the *Object Constraint Language (OCL)* enrich the abstract syntax, aiming to the validation of models. The meta-model has been implemented using the *Eclipse Modeling Framework (EMF)*, which provides powerful modeling tools for these purposes. In the remainder of this section, an introduction to the M4J-DSL will be given. Section 3.1.1 gives an overview of the underlying meta-model. Section 3.1.2 illustrates of some related testing features supported by Eclipse.

3.1.1 Overview of the Meta-Model

Figure 3.2 shows the meta-model which specifies the entities of the Markov4JMeter workload generation model and their relations. Additionally, a total of 34 OCL constraints (listed in Appendix A) have been identified and formulated. With regard to the Markov4JMeter workload model, the constraints ensure that M4J-DSL models which comply to the depicted meta-model do not have invalid characteristics beyond that, e.g., negative values for relative frequencies, multiple definitions of the same service names, or Markov chain transitions which do not correspond to the Session Layer. The constraints have been implemented by the use of the OCLInEcore utilities as introduced Section 2.4.2. The remainder of this section gives a more detailed description of individual classes and coherent parts.

- ▷ The class *WorkloadModel* denotes the root of a workload model. It consists of a workload intensity, an Application Model, the Behavior Mix and a set of Behavior Models including at least one instance. The names of the corresponding classes indicate these parts.
- ▷ The workload intensity needs to be provided as a formula for the Session Arrival Controller of the Markov4JMeter add-on for JMeter. Hence, the class *WorkloadIntensity* contains a related attribute. The class is declared as abstract, serving as a base class for miscellaneous types of workload intensity. Type-specific subclasses can be derived to provide additional attributes, for example a constant number of sessions as defined in class *ConstantWorkloadIntensity*. This might be useful for future purposes, e.g., evaluating a formula without parsing its string representation. However, the Session Arrival Controller just needs the formula string, since it parses the string itself.
- ▷ The class *ApplicationModel* represents a hierarchical finite state machine including an EFSM for the Session Layer and EFSMs for the Protocol Layer, as defined by van Hoorn et al. [2008]. This class neither contains any further attributes nor does it provide any additional information except the Session Layer EFSM, but it has been added for illustration purposes and possible enhancements.

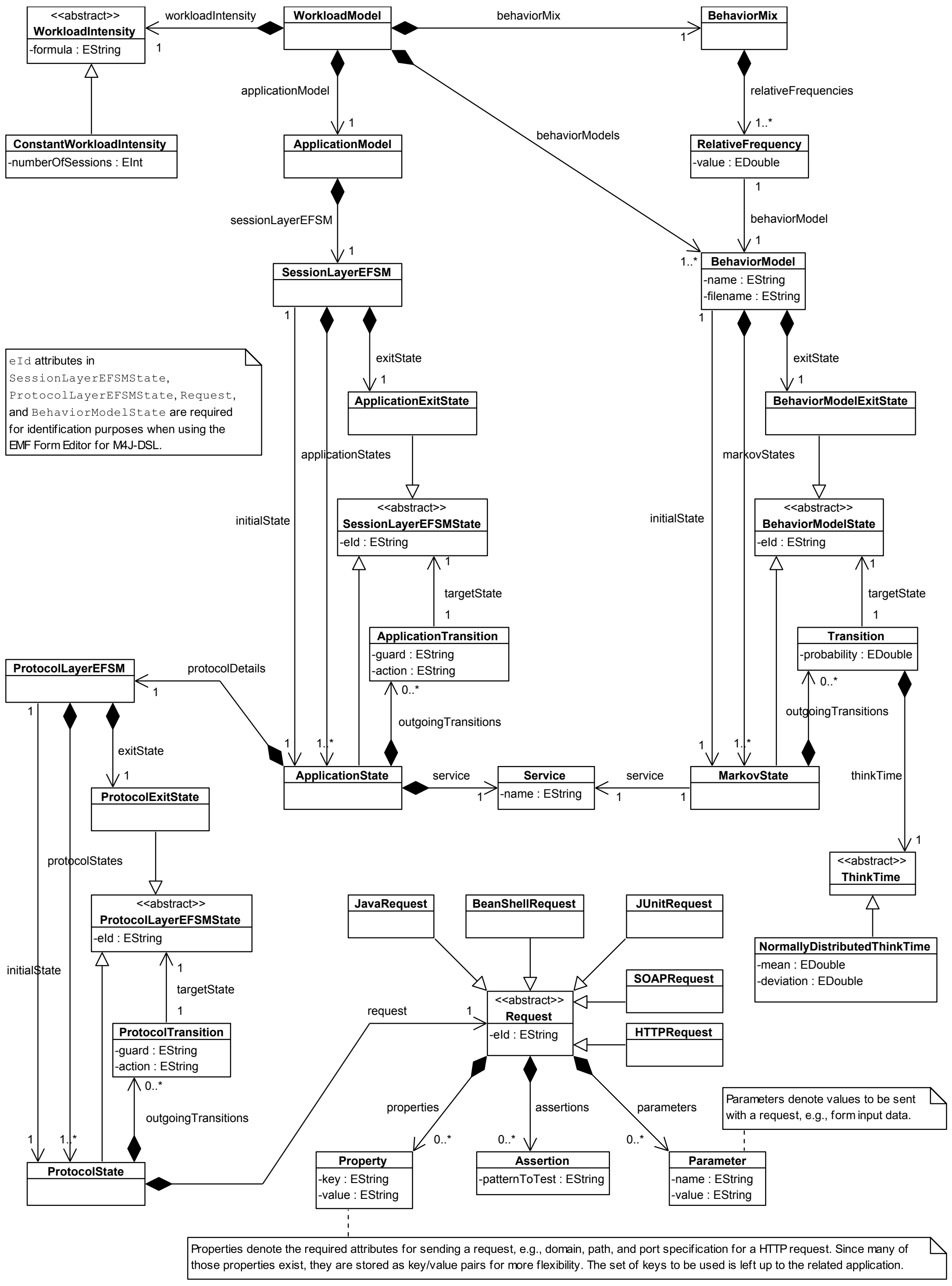


Figure 3.2. Meta-model for Markov4JMeter workload models. Related OCL constraints are listed in Appendix A.

▷ A Session Layer EFSM consists of states and transitions, with each state representing either an Application State associated with a service, or a dedicated exit state. Each Session Layer EFSM has at least an initial state and an exit state as well. This issue is modeled by class *SessionLayEFSM* and its associated state classes. The exit state of a Session Layer EFSM is not associated with any service, since no service is called when the exit state is reached. Consequently, it is modeled as a dedicated state of the Session Layer EFSM. In contrast, an initial state is defined by van Hoorn et al. [2008] as being one of the standard Application States. Transitions between states include guards and actions as strings to be evaluated by the Markov Session Controller of the Markov4JMeter add-on.

▷ An instance of class *Service* has a (unique) name for identification purposes. It is important to note that states and services do not coincide, since states denote structural elements, and services do not.

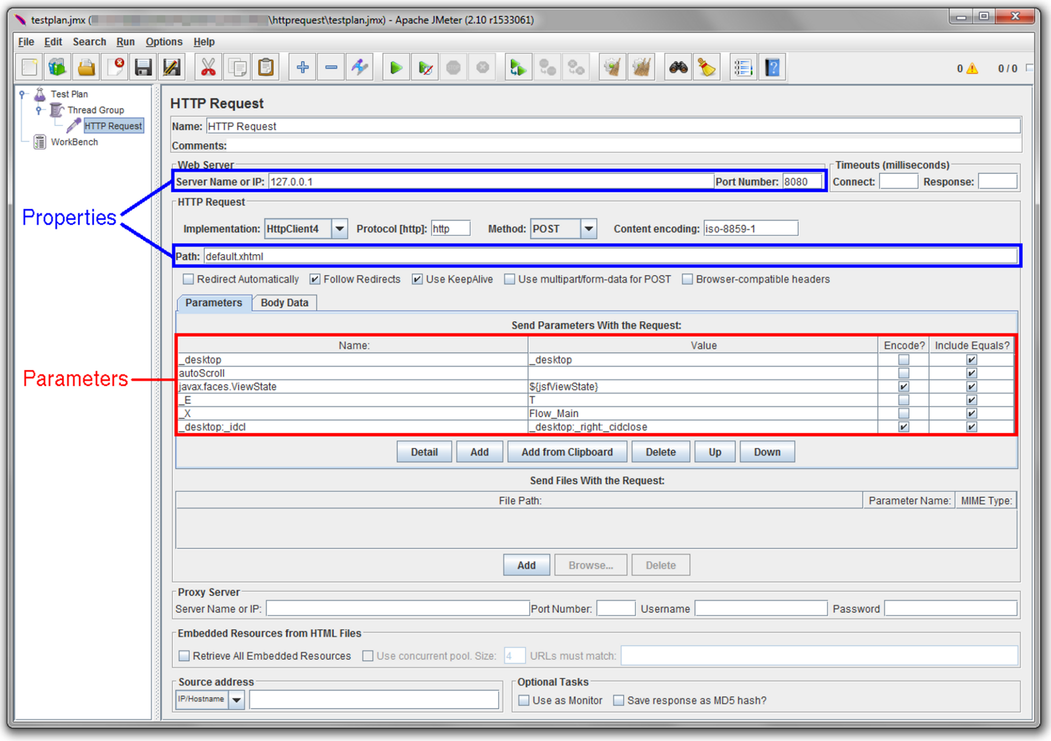
▷ Each Application State is associated with an EFSM of the Protocol Layer. The structure of a Protocol Layer EFSM principally corresponds to the structure of a Session Layer EFSM. However, each non-exit Protocol State is associated with a request instead with a service.

▷ *Request* is the abstract base class for any type of requests, which differ in their related protocol, e.g., HTTP, SOAP, or Java requests. Each request contains properties and parameters, which can be classified as follows:

- Properties denote the required attributes for sending a request, e.g., domain, path, and port specification for an HTTP request. Figure 3.3 shows a screenshot of a JMeter configuration panel for HTTP requests, including several blue marked example values. Since many of those properties exist, they are stored as key/values pairs for more flexibility. Even non-protocol specific information like a name or a comment might be specified as property. The set of keys to be used is left up to the related application.
- Parameters denote values to be sent with a request, e.g., form input data for a Web application. Figure 3.3 includes several red marked example parameters. These are stored as name/values pairs, whereas the names must be unique within a request, analogous to property keys.

Optionally, a request even contains assertion patterns which might be applied to the regarding response data. It is important to note that the M4J-DSL is not JMeter-specific and that Figure 3.3 serves for clarification purposes only.

▷ The *BehaviorMix* class contains a set of relative frequencies which are assigned to Behavior Models. At least one relative frequency must exist, to ensure that a frequency sum of 1 is possible.



HTTP Request

Name: HTTP Request

Comments:

Web Server

Server Name or IP: 127.0.0.1 Port Number: 8080

Timeouts (milliseconds)

Connect: Response:

HTTP Request

Implementation: HttpClient4 Protocol [http]: http Method: POST Content encoding: iso-8859-1

Path: default.xhtml

☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data for POST ☐ Browser-compatible headers

Send Parameters With the Request:

Name:	Value	Encode?	Include Equals?
_desktop	_desktop	☐	☑
autoScroll		☐	☑
javax.faces.ViewState	${jsfViewState}	☑	☑
_E	T	☐	☑
_X	Flow_Main	☐	☑
_desktop:_idcl	_desktop:_right:_cidclose	☑	☑

Send Files With the Request:

File Path:	Parameter Name:	MIME Type:

Proxy Server

Server Name or IP: Port Number: Username Password

Embedded Resources from HTML Files

☐ Retrieve All Embedded Resources ☐ Use concurrent pool. Size: 4 URLs must match:

Source address

IP/Hostname

Optional Tasks

☐ Use as Monitor ☐ Save response as MD5 hash?

Figure 3.3. JMeter configuration panel for an HTTP request, illustrating request properties (domain, port, and sub-path) as well as several parameters by example.

▻ The *BehaviorModel* class represents a Markov chain, including two attributes: a (unique) name for identification purposes and a (unique) filename which determines the location of the CSV file containing a probability matrix. The model structure of a Markov chain is similar to the Session Layer EFSM structure. However, an outgoing transition of a Markov State contains a probability value and an instance of a *ThinkTime* subclass.

▻ *ThinkTime* is the abstract base class for any type of think times, differing in the related distribution, e.g., binomial or normal distribution. Each think time is associated with a transition of a Markov chain. A subclass needs to provide the required attributes for appropriate delays being computed in an experiment.

▻ As indicated by its name, class *NormallyDistributedThinkTime* represents a think time which depends on a normal distribution. It provides a mean and a standard deviation value for the computation of appropriate delays.

Several classes include an additional attribute named *eId*, for being identified when using certain types of EMF editors, e.g., the so-called *Generic EMF Form Editor*. It is important to note that these attributes, shortly referred to as *identifiers*, have been added for that purpose only. They do not provide any additional semantic information to the workload model itself.

The meta-model has been designed with regard to flexibility and extensibility. Advanced workload intensities, additional think time distributions, or even new request types can be simply added as new subclasses of their related abstract base classes. Currently supported request types respectively protocols are Java, BeanShell, JUnit, SOAP, and HTTP. The next section describes how example instances of the M4J-DSL can be created for testing purposes.

3.1.2 Tool Support

Eclipse offers the option to create *dynamic* model instances which can be used for testing purposes in particular. That is, instances of any (non-abstract) model classes can be constructed via EMF Form Editors and stored as serialized objects in *XML Metadata Interchange (XMI)* files for being reused in editors or even in Java programs as for the transformations described later. EMF additionally supports corresponding libraries for processing such files easily and deserializing their XMI content back to Ecore objects. Furthermore, EMF Form Editors allow the construction of model instances including an optional "live validation", which means that any defined constraints are evaluated while the model is being built by the user. Figure 3.4 shows an excerpt of the Generic EMF Form Editor in Eclipse with an opened pop-up menu offering the option to create a dynamic instance of the *WorkloadModel* class. It also depicts the activated *Live Validation* option.

During the construction process, the (live) validator traverses recursively the tree structure of a model for detecting any problems in the individual nodes. Problems will be displayed in the regarding node and all its parents as well. Each error message includes the name of a violated constraint. Figure 3.5 shows the *Generic EMF Form Editor* with an opened XMI file containing a model which does not comply to the M4J-DSL. Assume that the Application State *AS_Login* does not contain a self-transition in the Session Layer EFSM of the given model, meaning that it must not have an outgoing Application Transition with *AS_Login* as target state. Adding such an outgoing transition to the corresponding Markov State *MS_Login* causes the editor to display initially the two depicted problems: since self-transitions in *AS_Login* are not allowed, the *mustBeOutgoingTransitionsCorrespondingToSessionLayer* constraint is violated in Markov State *MS_Login*. The second issue results from the undefined think time of the (invalid) transition. It is recommended to resolve such problems from the affected most inner nodes of the tree, to keep a clear overview on outstanding issues. The live validation might take several seconds of time for large models, that is models with hundreds or thousands of states. Consequently, (error-)messages might be updated strongly delayed, if any model changes are made. However, the live validation helps to identify errors in a simple and clear way.

The benefits of using the Ecore OCL features are obvious: the validation code for a M4J-DSL model is generated straight from the Ecore model, and therewith the validation

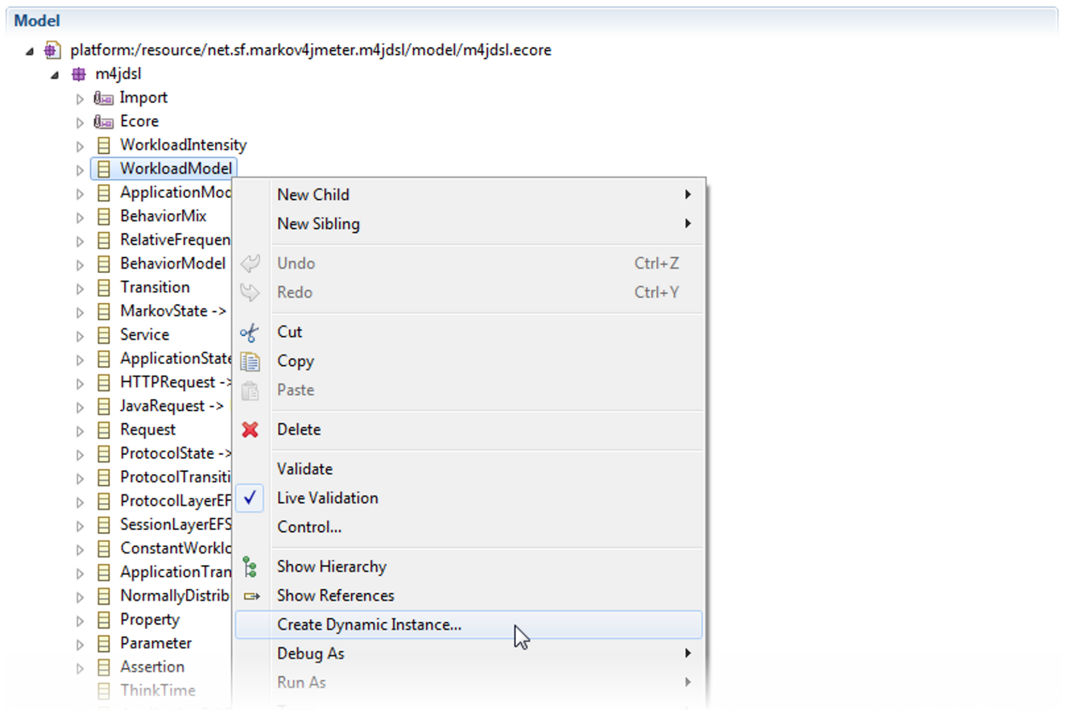


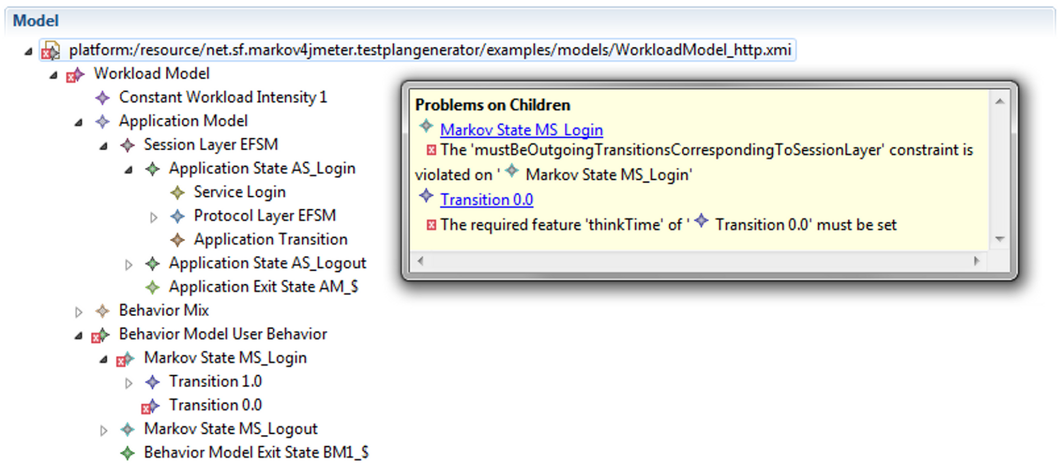
Figure 3.4. *EMF Generic Form Editor* (excerpt) with the option to create a dynamic model instance.

can be implemented without writing manual (Java) code, reducing the risk of errors and saving lots of time.

3.2 Test Plan Generation Framework

The Test Plan Generation Framework, shortly referred to as *Test Plan Generator*, builds for a given M4J-DSL model a corresponding Test Plan which can be loaded and executed in JMeter. It provides the validation of input models to avoid invalid output Test Plans. Generated Test Plans have a uniform core structure, depending on the implementation of the generator's transformation unit. Additionally, so-called *filters* can be applied to any resulting Test Plan for modifying its uniform core structure. The architecture offers a high degree of extensibility with respect to alternative Test Plan structures and to additional functionality as well. The generator has been implemented as a standalone application in Java and is executable from command-line.

Section 3.2.1 gives a brief overview of the essential process performed by the generator when it transforms an M4J-DSL model into a JMeter Test Plan. Section 3.2.2 describes the



```
Model
platform:/resource/net.sf.markov4jmeter.testplangenerator/examples/models/WorkloadModel_http.xmi
  Workload Model
    Constant Workload Intensity 1
    Application Model
      Session Layer EFSM
        Application State AS_Login
          Service Login
          Protocol Layer EFSM
          Application Transition
        Application State AS_Logout
        Application Exit State AM_$
    Behavior Mix
    Behavior Model User Behavior
      Markov State MS_Login
        Transition 1.0
        Transition 0.0
      Markov State MS_Logout
      Behavior Model Exit State BM1_$
```

```
Problems on Children
Markov State MS_Login
The 'mustBeOutgoingTransitionsCorrespondingToSessionLayer' constraint is violated on ' Markov State MS_Login'
Transition 0.0
The required feature 'thinkTime' of ' Transition 0.0' must be set
```

Figure 3.5. *Generic EMF Form Editor* (excerpt) with an opened XMI file, displaying several model problems caused by a violated OCL constraint and undefined properties.

usage of the Test Plan Generator, including a more detailed introduction to its features. Section 3.2.3 outlines the architecture of the framework, illustrating its package and class structure. Section 3.2.4 describes the extensibility of the framework in terms of new features and additional Test Plan elements. Section 3.2.5 conclusively discusses limitations and open issues for future work.

3.2.1 Overview

As illustrated in Figure 3.1, the primary task of the Test Plan Generator is the transformation of a given M4J-DSL input model, provided as (optionally XMI-serialized) Ecore object, into a JMeter Test Plan. The transformation process is based on a set of configuration properties and Test Plan default properties as well. Test Plan elements share a single configuration file which additionally references CSV files for default parameter values. The transformation output consists of a JMX file, denoting a Test Plan in XML format which can be loaded into the JMeter tool. Additionally, the Behavior Models contained in the input model are written to CSV files, referenced by the output Test Plan. Optionally, an "out of the box" test run using the JMeter engine with a newly generated Test Plan might be started straight from the Test Plan Generator afterwards, but this is still work in progress.

Figure 3.6 shows the internal input-output transformation process of the Test Plan Generator. The *Generator Initialization* task includes several configuration steps, e.g., parsing input parameters, organizing the output file paths, and setting up the JMeter engine. After the setup, the input model is validated. In case the validation fails, the generator does not produce any output files and gives an error message instead, indicating the validation error which occurred. If the validation is successful, the *Model Transformation* task is started. Objects of the given M4J-DSL model are mapped to corresponding Test Plan

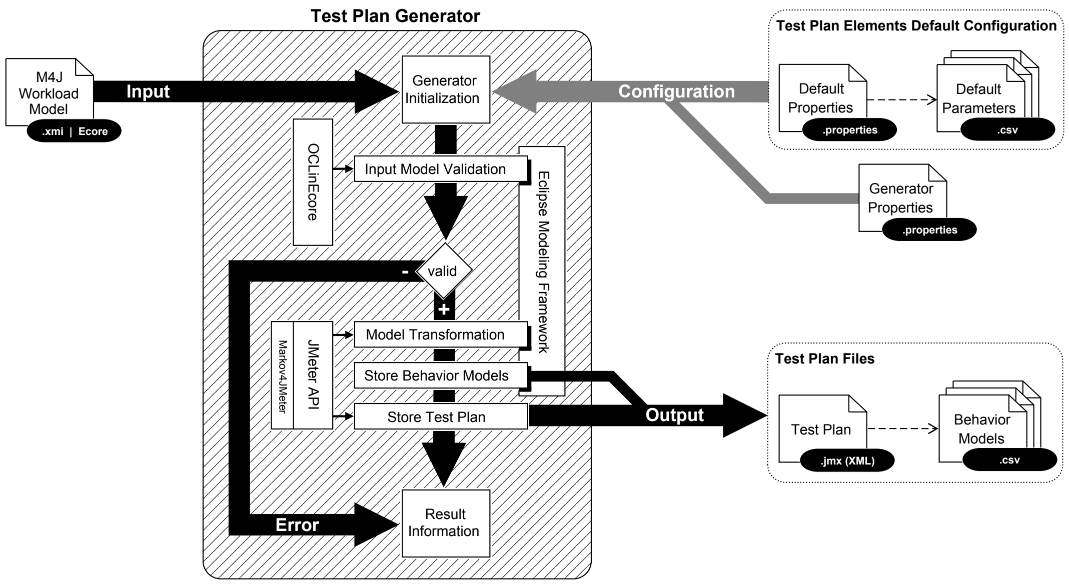


Figure 3.6. Input-output transformation process of the Test Plan Generator.

fragments. Afterward, the Behavior Models are written to CSV files, located in a single output folder for being referenced by the Test Plan which is written subsequently. Finally, a result information will be given, indicating possible failure (e.g., if file access fails) or success.

The validation of input models is based on OCLinEcore utilities which have been introduced in Section 2.4.2. It includes the evaluation of those constraints which have been discussed in context of the M4J-DSL in Section 3.1. Furthermore, the *Model Transformation* task makes use of the JMeter API which provides the Test Plan elements as well as container classes for the output structure. Additionally, the JMeter API provides the serialization of the Test Plan object structure into an XML script, required in the *Store Test Plan* task. The EMF is the underlying core technology for the previous tasks.

3.2.2 Usage

The Test Plan Generator is a standalone application which has been deployed as a single JAR file, to be executed from command-line or included as external library into other Java projects. In case the application is executed from command-line, the input model has to be passed as an XMI file, containing a serialized Ecore model. Such file might be obtained by saving an M4J-DSL model instance in an EMF Form Editor, as described in Section 3.1.2; the M4J-DSL instance which results from generative application data, as discussed later in Section 4.1, will be stored as XMI file, too. When being used as library, the generator might be even started with an Ecore object passed as input; for that case, refer to Section 3.2.3 which discusses the class details.

Currently, the framework supports a suitably selected subset of JMeter Test Plan elements, completely pre-configurable with default properties. An overview of these elements is given in Appendix B. In particular, a certain set of request types, that is corresponding JMeter Samplers, is supported. An M4J-DSL input model must use a unique request type only. The used type will be detected by the framework automatically.

For optimization purposes, filters might be applied to any Test Plan which results from the core transformation process. Each filter implements additional modifications to the Test Plan; for example, a filter might collect common HTTP request values in a Test Plan and move them into a global *HTTP Request Default Values* element for making manual changes more comfortable. Filters to be applied can be specified as a sequence of their short names, in an arbitrary combination and in the order in which they shall be applied. However, there are still some open issues with regard to filters left, which will be discussed in Section 3.2.5.

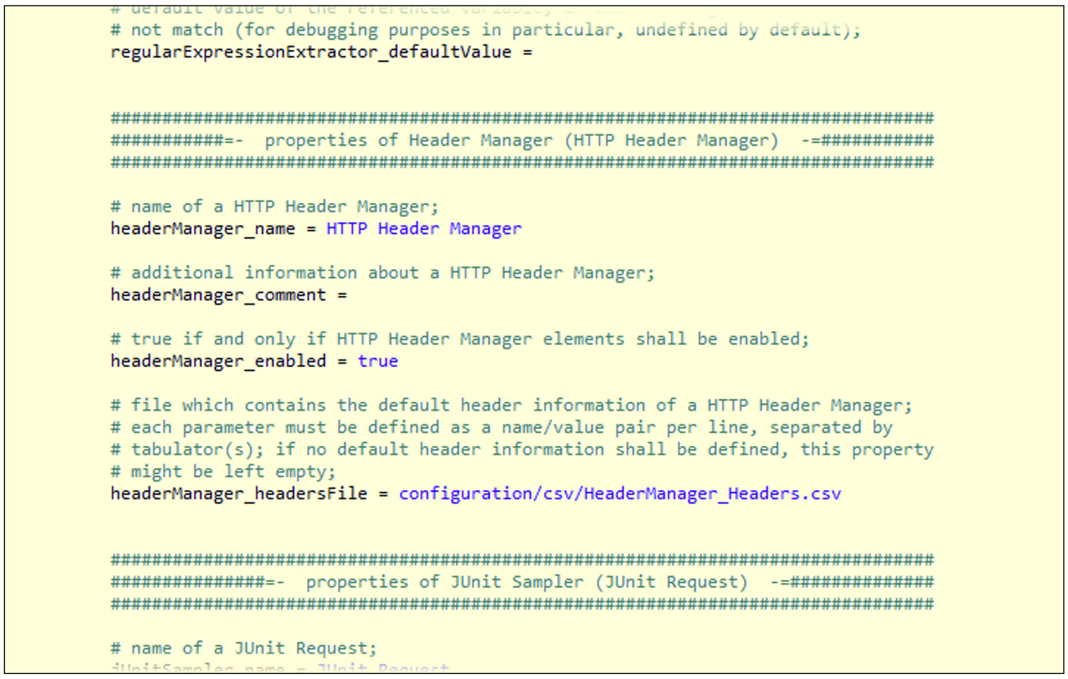
Configuration

As illustrated by Figure 3.6, the Test Plan Generator configuration consists of generator-specific properties and a default configuration for Test Plan elements. Both information must be provided as properties files and (optional) CSV files as well. The generator-specific configuration contains the locality and further settings for the JMeter engine; appropriate default settings are provided and should be sufficient, making changes unnecessary in general. In contrast, the default settings for Test Plan elements must be specified explicitly. They have to be defined in a single properties file, whose path must be passed as a command-line parameter, or as a method parameter if the transformation is executed programmatically. Figure 3.7 depicts an excerpt of such a file, showing the definition of default properties for a Test Plan element named *HTTP Header Manager*.

Besides their properties like name, comment, and activity status, certain Test Plan elements also have parameters to be defined. For example, the *HTTP Header Manager* whose JMeter configuration panel is depicted in Figure 3.8, requires header information to be sent with each HTTP request. Such parameters can be defined in CSV files to be read by the generator. The related CSV file paths must be registered in the main properties file. Whenever an instance of the regarding Test Plan element needs to be built, its default parameters will be read from the registered file. This allows the instantiation of fully customized Test Plan elements. Figure 3.9 shows the content of a CSV file for the *HTTP Header Manager* with parameter definitions. These parameters are reflected in the input form depicted in Figure 3.8.

Command-Line Parameters

The Test Plan Generator accepts a set of command-line parameters for configuration purposes, with each parameter being specified with a leading hyphen; an overview is given in Table 3.1, and several examples will be discussed next.



```
# not match (for debugging purposes in particular, undefined by default);
regularExpressionExtractor_defaultValue =


################################################################################
###########=-  properties of Header Manager (HTTP Header Manager)  -=###########
################################################################################

# name of a HTTP Header Manager;
headerManager_name = HTTP Header Manager

# additional information about a HTTP Header Manager;
headerManager_comment =

# true if and only if HTTP Header Manager elements shall be enabled;
headerManager_enabled = true

# file which contains the default header information of a HTTP Header Manager;
# each parameter must be defined as a name/value pair per line, separated by
# tabulator(s); if no default header information shall be defined, this property
# might be left empty;
headerManager_headersFile = configuration/csv/HeaderManager_Headers.csv


################################################################################
###############=-  properties of JUnit Sampler (JUnit Request)  -=##############
################################################################################

# name of a JUnit Request;
```

Figure 3.7. Properties file for Test Plan elements (excerpt), illustrating the definition of default properties for an *HTTP Header Manager*.

Example Options

▻ The option sequence

```
-i WorkloadModel.xmi -o testplan.jmx -t testplan.properties
```

denotes a minimum start configuration for the Test Plan Generator, defining the files "WorkloadModel.xmi" and "testplan.jmx" to be used as input file and output file respectively, and it directs the generator to use the default values provided by file "testplan.properties" for Test Plan elements.

▻ The option sequence

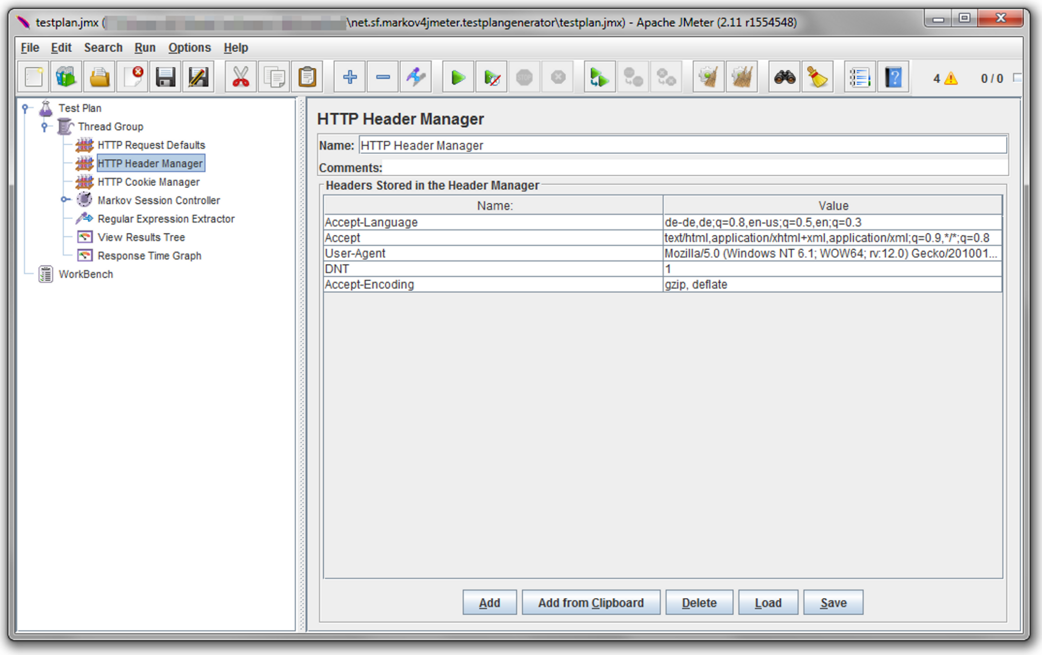
```
-i WorkloadModel.xmi -o testplan.jmx -t testplan.properties -l 2
```

has the same effect as the first one, but it additionally defines a MacOS-specific line-break type to be used for the CSV files of the Behavior Models.

▻ The option sequence

```
-i WorkloadModel.xmi -o testplan.jmx -t testplan.properties -l 2 -r
```

has the same effect as the second one, but it additionally uses the -r option for starting a test run with the resulting Test Plan in the JMeter engine immediately.



HTTP Header Manager

Name: HTTP Header Manager

Comments:

Headers Stored in the Header Manager

Name:	Value
Accept-Language	de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0) Gecko/201001...
DNT	1
Accept-Encoding	gzip, deflate

Figure 3.8. JMeter configuration panel for an *HTTP Header Manager* element, illustrating parameter options of a Test Plan element.

Input Model Validation

Models passed as input to the Test Plan Generator must comply to the M4J-DSL specification. Hence, the framework initially validates the OCL constraints defined in Section 3.1 on a given input model. If the validation fails, an error message regarding violated constraints will be given. Table A.1 in Appendix A gives an overview of the possible error messages. The transformation process will start if and only if the validation of a given input model is successful.

```
Accept-Language     de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept              text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent          Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0) Gecko/20100101 Firefox/12.0
DNT                 1
Accept-Encoding     gzip, deflate
```

Figure 3.9. CSV file with default parameter definitions for a Test Plan element (*HTTP Header Manager*), using tabulators as separators between keys and values. This is the content of the file assigned to the key `headerManager_headersFile` in the excerpt shown by Figure 3.7.

Table 3.1. Test Plan Generator command-line options. *Long* denotes a long option name, and *Short* denotes its related flag.

Long	Short	Description
`input`	`i`	XMI input file which provides the M4J-DSL workload model to be transformed into a JMeter Test Plan, e.g., "Workload-Model.xmi".
`output`	`o`	Output file of the JMeter Test Plan; the suffix ".jmx" indicates a JMeter Test Plan file, e.g., "testplan.jmx".
`testplanproperties`	`t`	Properties file which provides the default values of the Test Plan elements.
Optional Arguments		
`linebreak`	`l`	(Optional) OS-specific line-break for being used in the CSV files of the Behavior Models (0 = Windows, 1 = Unix, 2 = MacOS); the default value is 0 (Windows).
`path`	`p`	(Optional) path to an existing destination directory for the Behavior Model files to be written into; the default value is "./" (current directory).
`generatorproperties`	`g`	(Optional) properties file which provides the configuration values of the Test Plan Generator, in particular the locality and further settings for the JMeter engine; the default settings should be sufficient, so that this option does not need to be used in general.
`filters`	`f`	(Optional) filters for being applied to the resulting Test Plan after the transformation process; filters must be passed as a sequence of their short names, in an arbitrary combination and order.
`runtest`	`r`	(Optional) immediate start of the JMeter engine for running a test with the resulting Test Plan; this option has no additional argument.

Transformation Process

In the transformation process, objects of a valid M4J-DSL model are mapped to corresponding Test Plan fragments. Required Test Plan elements are created with the use of the JMeter API which additionally provides container classes for the output structure.

The M4J-DSL input model supplies information about the workload intensity, the Application Model, the Behavior Mix, and the Behavior Models. All of this information must be arranged properly in the resulting Test Plan. Figure 3.10 illustrates the mapping of M4J-DSL objects to corresponding Test Plan fragments by example. The underlying model describes a simple application which only allows the user to sign in and sign out. The Markov chain depicted in Figure 3.11 models some regarding user behavior with example probability values; think times have been excluded for the sake of clarity. It is assumed that the Session Layer EFSM does not define any transitions beyond those of the given Markov chain. The M4J-DSL instance whose structure is shown on the left side of Figure 3.10 represents that EFSM. Application States named *AS_Login* and *AS_Logout* are associated

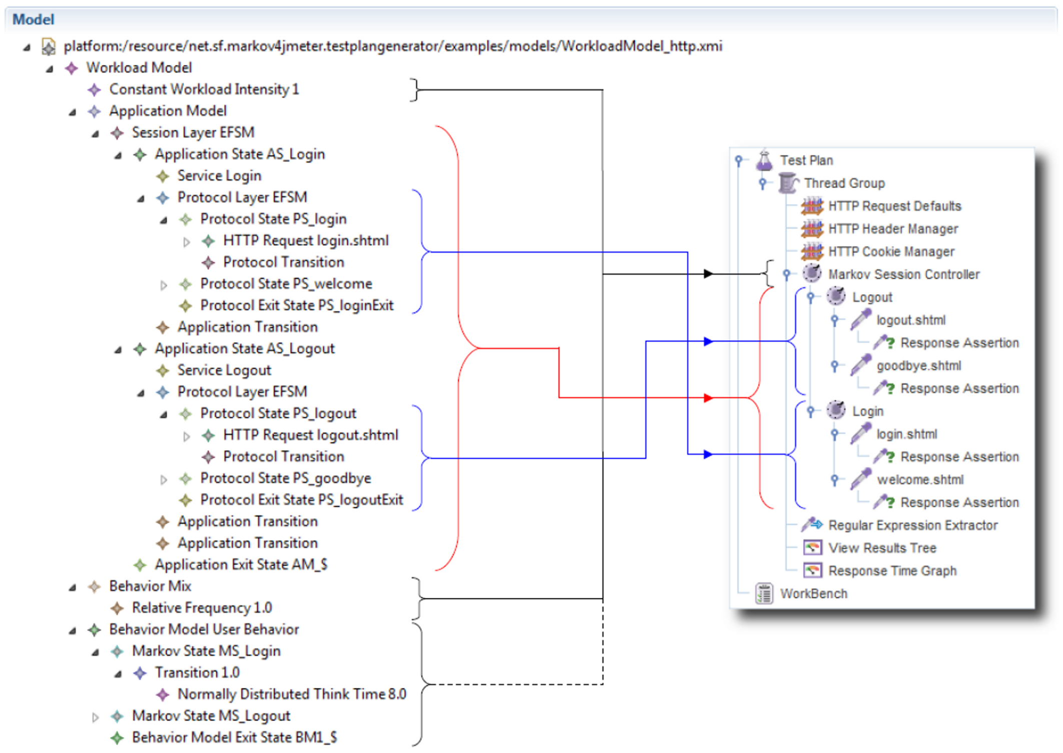


Figure 3.10. Mapping of M4J-DSL objects to corresponding JMeter Test Plan fragments.

with services named *Login* and *Logout* respectively, as well as some related Protocol Layer EFSMs. Corresponding Markov States are denoted as *MS_Login* and *MS_Logout*, and transitions between the states are modeled accordingly. A constant workload intensity of 1 is defined, and the Behavior Mix contains a relative frequency of 1.0 for the – only existing – Behavior Model.

▻ Black braces comprise the information of the M4J-DSL model which is put into the Markov Session Controller of a resulting Test Plan, as depicted on the right side of Figure 3.10: workload intensity, Behavior Mix, and Behavior Models. The latter is indirectly – indicated by a dashed line – stored in the Markov Session Controller, since Behavior Models are put into separate CSV files which are referenced by the controller's

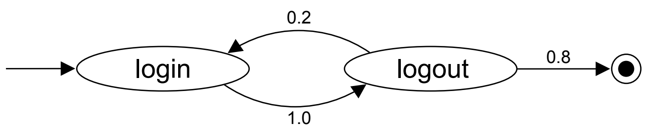


Figure 3.11. Markov chain modeling user behavior regarding to a simple login/logout application.

Behavior Mix. In contrast, Behavior Mix and workload intensity are written directly into the Markov Session Controller and its Session Arrival Controller, respectively.

▷ Red braces illustrate the mapping between the Session Layer EFSM and its corresponding part of the Test Plan. The order of Markov States – note that Application States in Markov4JMeter are loosely identified with Markov States of Behavior Models – in the JMeter Test Plan does not comply to the order of their corresponding Application States in the M4J-DSL model; this results from the algorithm which is used for transformation, since states are traversed in *depth-first search (DFS)* order and "deepest" states are visited at first. However, state order does not matter here, since the selection of states is determined by the Markov Session Controller only.

▷ Blue braces indicate the Protocol Layer mapping. It is important to note that currently no conditional branches are supported for Protocol Layer EFSMs. This results from the non-availability of *GOTO* Test Plan elements in JMeter, which makes branching to specific target states difficult. This will be discussed as an open issue in Section 3.2.5.

Result Output

As indicated by Figure 3.6, the generator output consists of a Test Plan and a set of Behavior Models. The Test Plan will be written as JMX file, denoting JMeter content, into a given output file path. When the generator is executed via command-line, the path defined by the *output* option as specified in Table 3.1 will be used. Behavior Models will be stored as CSV files in the directory defined by the *path* option; the denoted folder must already exist; it will not be created. In case the option is not used, the current directory will be selected. It is important to note that the output location of Behavior Models is *final*, that is moving the files to any other location will dissolve the binding between the Test Plan file and the Behavior Models. The reason is that the Session Controller of the Markov4JMeter add-on currently does not accept relative file paths; in consequence, absolute file paths are stored in Test Plans. For changing the location of Behavior Models, the generation process must be restarted with a customized output path.

3.2.3 Architecture

This section gives an overview of the Test Plan Generator's architecture. It starts with an introduction of the package structure, illustrating the core units. Afterward, an outline of the class structure will be given, discussing the main classes in more details. The section closes with an overview of external libraries used for the framework. It is important to note that the description does not cover all code details; those details should be obtained from the source code itself, the Java code is comprehensively commented.

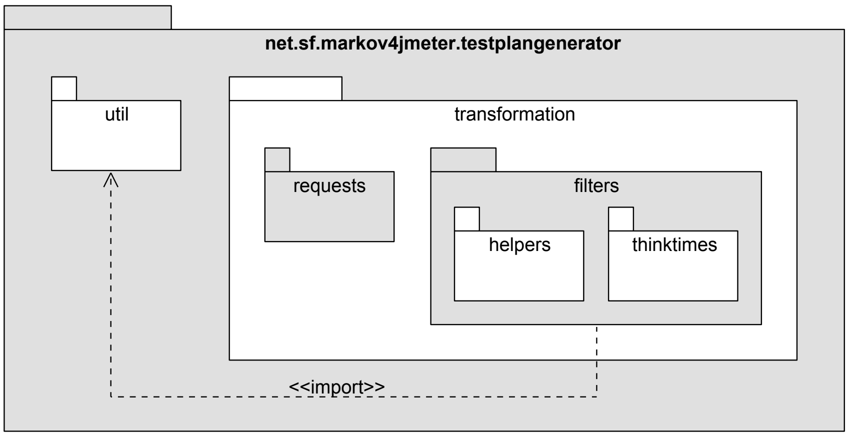


Figure 3.12. Package structure of the Test Plan Generator

Package Structure

The packages of the Test Plan Generator are organized in a mainly hierarchical order. Figure 3.12 gives an overview of the structure, depicting the nested packages which are included in the *net.sf.markov4jmeter.testplangenerator* root package. The content of the individual (sub-)packages is as follows:

▷ The *util* package provides general-purpose classes for simplifying miscellaneous base tasks. These classes are not framework-specific and have been designed for also being integrated into any other system in a simple and efficient way. In particular, classes for system configuration, CSV file handling, Ecore objects validation, and (de-)serialization of Ecore objects are provided.

▷ The *transformation* package includes the core classes for the model-to-model (M2M) transformation process, which converts a given M4J-DSL model into a corresponding JMeter Test Plan structure. The package particularly contains the base code for building appropriate Test Plan fragments for the EFSMs of the Session Layer and the Protocol Layer of a Markov4JMeter model. Further transformation classes are divided into sub-packages.

▷ The *transformation.requests* package contains classes for transforming certain types of requests into corresponding JMeter Samplers. Currently supported protocols are *HTTP* and *SOAP*, and Java method invocation is supported through *Java*, *BeanShell*, and *JUnit* Samplers. Java method invocation via JMeter can be done as discussed in Section 2.7.

▷ The *transformation.filters* package provides the code for filters to be applied on Test Plans after the main transformation process, as described in Section 3.2.2. The installation of the Behavior Mix into a Test Plan, including the writing of CSV files with Behavior

Models, is also implemented as a filter. Hence, the filter package uses classes of package *util* for writing these CSV files. A dedicated sub-package for the think time classes has been located in the filters package for keeping the hierarchical order. Furthermore, a sub-package with helper classes for Test Plan modification is included.

▻ The *transformation.filters.thinktimes* package contains formatter classes for think times being stored in Behavior Model matrices. In particular, the package includes an abstract base class from which think time specific classes must be inherited. The currently supported think time type depends on the Gaussian distribution, alternative think time types can be simply added. This will be discussed in Section 3.2.4.

▻ The *transformation.filters.helpers* package provides helper classes for Test Plan modification. These classes simplify the search for specific Test Plan elements, insertion of new elements at certain positions, or even removal or replacement of elements with others.

Class Structure

The class model of the Test Plan Generator consists of around 30 classes. For the sake of clarity, its structure will be divided into three essential parts when being introduced in this section. The first part concerns the classes required for initialization and result output purposes. The second and third parts concern the transformation process and the filters implementation respectively. It is important to note that the attributes and operations of the class diagrams depicted in this section are not intended to be complete; they serve for illustration purposes only. In particular, the signatures of operations must not necessarily comply to the signatures in the source code, to avoid confusion caused by large parameter lists.

Initialization and Result Output When the Test Plan Generator is executed, a set of parameters, including, for example, the M4J-DSL input model and an output path, is read from command-line. The input model needs to be read and validated, before the transformation process can start. For transforming the input model to a JMeter Test Plan, the generator uses a factory for creating appropriate JMeter Test Plan elements. This factory needs to be initialized under the use of a default configuration. For certain Test Plan elements, the factory might read additional default parameters from CSV files. A generated Test Plan is written as XMI file to the specified output path. Optionally, the JMeter Engine might be started with the generated Test Plan being passed as input parameter for an immediate test run.

All of these tasks have been encapsulated into individual classes which are illustrated including their relations to each other in Figure 3.13. The diagram mainly includes classes of the *util* package, namely *Configuration*, *CSVHandler*, *EcoreObjectValidator*, and *XmiEcoreHandler*. Remaining classes are located in the root package of the framework.

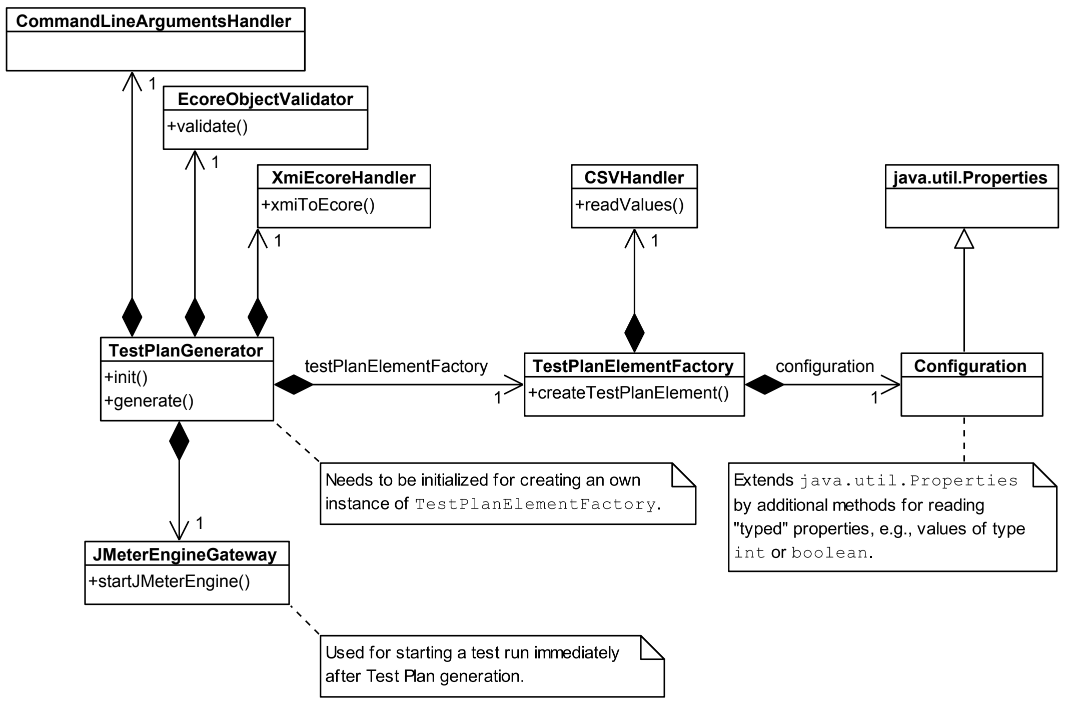

CommandLineArgumentsHandler
EcoreObjectValidator
+validate()
XmiEcoreHandler
+xmiToEcore()
CSVHandler
+readValues()
java.util.Properties
TestPlanGenerator
+init()
+generate()
testPlanElementFactory
TestPlanElementFactory
+createTestPlanElement()
configuration
Configuration
JMeterEngineGateway
+startJMeterEngine()
1
Needs to be initialized for creating an own instance of TestPlanElementFactory.
Extends java.util.Properties by additional methods for reading "typed" properties, e.g., values of type int or boolean.
Used for starting a test run immediately after Test Plan generation.

Figure 3.13. Class model of the Test Plan Generator, concerning the initialization and result output.

▷ *TestPlanGenerator* is the base class which contains the `main()` method for standalone execution. Each instance of this class must be initialized by invoking a dedicated `init()` method; the initialization process comprises the creation of an associated factory for building Test Plan elements. The class even provides a `generate()` method for being invoked with either an Ecore object representing an M4J-DSL model or an XMI input file containing such a model to be loaded at first.

▷ *CommandLineArgumentsHandler* handles all available options to be passed via command line. Consequently, any new options must be defined in this class. This will be further discussed in Section 3.2.4.

▷ *EcoreObjectValidator* provides validation code for checking whether arbitrary Ecore objects follow their respective underlying meta-models. Besides the correctness of the core structure, OCL constraints (as discussed in Section 2.4.2) are evaluated by this class. Therefore, it makes use of a great amount of external libraries which are included in the framework. Additional notes will be given in the *External Libraries* part of this section.

▷ *XmiEcoreHandler* encapsulates the code for (de-)serialization of Ecore models. With this class, models can be read from XMI files and written back as well. The Test Plan Generator uses it for reading an input M4J-DSL model from an XMI file.

▷ *TestPlanElementFactory* provides methods for creating pre-configured JMeter Test Plan elements as described in Section 3.2.2. The class is associated with a configuration for obtaining default properties of such elements. Furthermore, it needs to read CSV files for retrieving default parameters of certain Test Plan elements. The elements are created by using the JMeter API; consequently, this class makes use of many external libraries provided by JMeter. For additional notes, refer to the *External Libraries* part of this section.

▷ *Configuration* extends the *java.util.Properties* class by additional methods for reading "typed" properties, e.g., values of type `int` or `boolean`. If a value is not available or invalid, its Java-specific default value is used.

▷ *CSVHandler* is used for handling the reading (and writing) of CSV files, e.g., the files which supply parameters as described in Section 3.2.2.

▷ *JMeterEngineGateway* serves as a gateway to the JMeter engine, which is used for running generated Test Plans immediately after they have been generated. There are still some open issues left for this task which will be discussed in Section 3.2.5.

Transformation The transformation of M4J-DSL models to JMeter Test Plans is mainly implemented through the classes of the *transformation* package. Figure 3.14 illustrates the corresponding class structure, indicating the hierarchical transformation process from Test Plan, Session Layer EFSM, and Protocol Layer EFSMs respectively, down to requests. In the following, only the main transformation classes will be discussed.

▷ *AbstractTestPlanTransformer* is the abstract base class for any test plan transformations to be implemented. Those implementations particularly define the core structure of the Test Plans to be built.

▷ *SimpleTestPlanTransformer* builds Test Plans which follow a minimum structure for being used with JMeter. In particular, it makes use of class *SimpleProtocolEFSMTransformer*, which will be discussed below.

▷ *SessionLayerEFSMTransformer* transforms the Session Layer EFSM of a given M4J-DSL into a resulting Test Plan fragment. The core transformation process remains always the same, except for the transformation of the EFSMs included in the Protocol Layer, which is done by a dedicated implementation of *AbstractProtocolLayerEFSMTransformer*.

▷ *AbstractProtocolLayerEFSMTransformer* is the abstract base class for any Protocol Layer EFSM transformer. Subclasses need to define the protocol type as well as the algorithm for traversing through a Protocol Layer EFSM. This follows the *strategy pattern* as

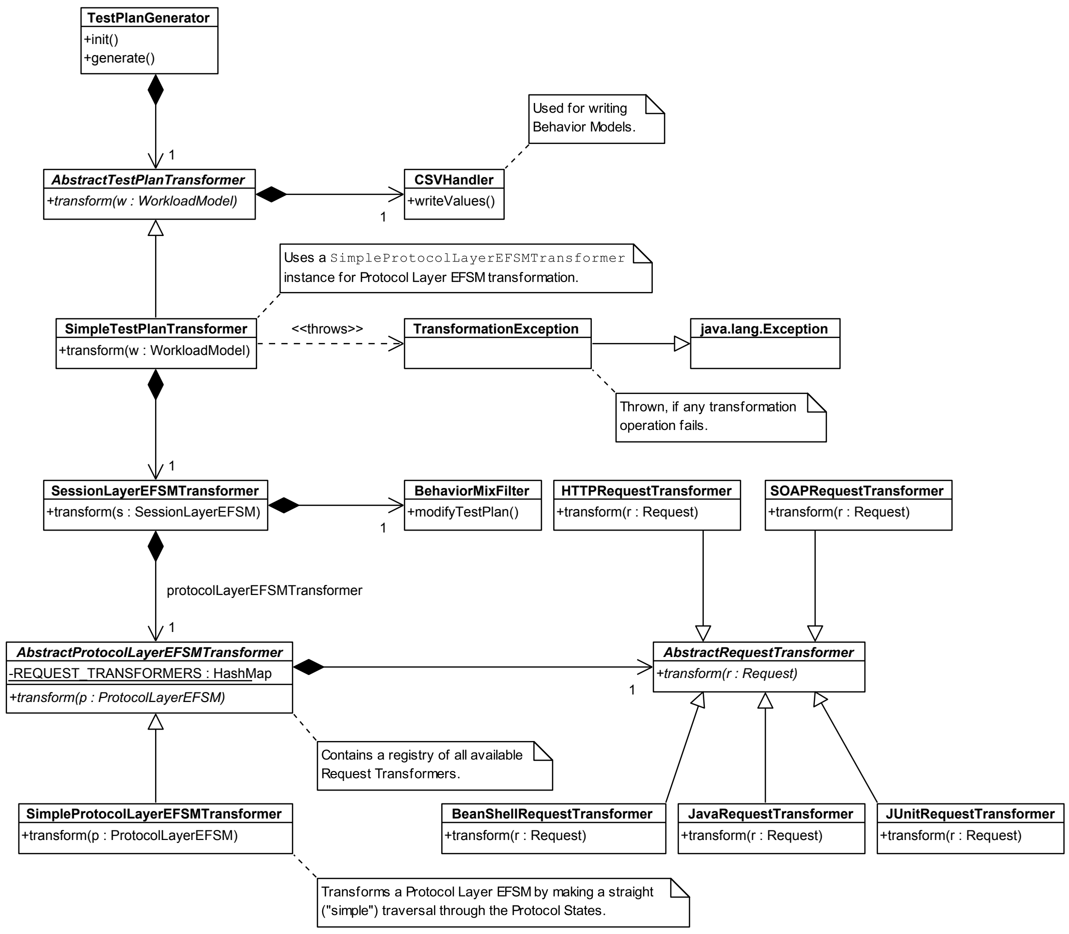

TestPlanGenerator
+init()
+generate()
Used for writing Behavior Models.
AbstractTestPlanTransformer
+transform(w : WorkloadModel)
CSVHandler
+writeValues()
1
Uses a SimpleProtocolLayerEFSMTransformer instance for Protocol Layer EFSM transformation.
SimpleTestPlanTransformer
+transform(w : WorkloadModel)
<<throws>>
TransformationException
java.lang.Exception
Thrown, if any transformation operation fails.
SessionLayerEFSMTransformer
+transform(s : SessionLayerEFSM)
BehaviorMixFilter
+modifyTestPlan()
HTTPRequestTransformer
+transform(r : Request)
SOAPRequestTransformer
+transform(r : Request)
protocolLayerEFSMTransformer
AbstractProtocolLayerEFSMTransformer
-REQUEST_TRANSFORMERS : HashMap
+transform(p : ProtocolLayerEFSM)
AbstractRequestTransformer
+transform(r : Request)
Contains a registry of all available Request Transformers.
SimpleProtocolLayerEFSMTransformer
+transform(p : ProtocolLayerEFSM)
BeanShellRequestTransformer
+transform(r : Request)
JavaRequestTransformer
+transform(r : Request)
JUnitRequestTransformer
+transform(r : Request)
Transforms a Protocol Layer EFSM by making a straight ("simple") traversal through the Protocol States.

Figure 3.14. Class model of the Test Plan Generator, concerning the transformation process; parameter types such as *WorkloadModel*, *SessionLayerEFSM*, and *Request* denote M4J-DSL classes and have not been depicted separately in this diagram for the sake of clarity.

introduced by Gamma et al. [1995, p.315-323], due to the fact that alternative solutions for a missing GOTO Sampler in JMeter must be found. This will be discussed as an open issue in Section 4.2.3.

▻ *SimpleProtocolEFSMTransformer* constitutes a partial solution for the non-availability of GOTO Test Plan elements in JMeter. Since no conditional branches are supported for Protocol Layer EFSMs currently, this transformer is specialized for a straight (“simple”) traversal through the Protocol States of a given Protocol Layer EFSM, assuming that no state has more than one outgoing transitions; if this restriction does not hold for a state, the traversal continues with the first outgoing transition, and a warning will be given.

▻ *AbstractRequestTransformer* is the abstract base class of any request type. currently supported requests are of type *HTTP*, *SOAP*, *BeanShell*, *Java*, and *JUnit* respectively.

Filters Similar to the *Pipes-and-Filters* pattern [Taylor et al., 2010, p. 110 – 111], filters can be chained, with a Test Plan being passed from one filter to the next. The Behavior Mix installation, including the storing of Behavior Models in CSV files, is implemented as a special filter which is applied to each Test Plan. Figure 3.15 shows the corresponding class structure, whose main classes will be discussed next.

▻ *AbstractFilter* is the base class of all filters. It provides an abstract *modifyTestPlan()* method which must by implemented by each subclass. An instance of *TestPlanGenerator* might handle an arbitrary number of filters, including at least an instance of *BehaviorMixFilter*.

▻ *HeaderDefaultsFilter* is intended to find common values in an HTTP-based Test Plan for storing them globally in an *HTTPHeaderDefaults* element. The search algorithm is not implemented yet; currently, this class serves as an example for the core structure of a filter class.

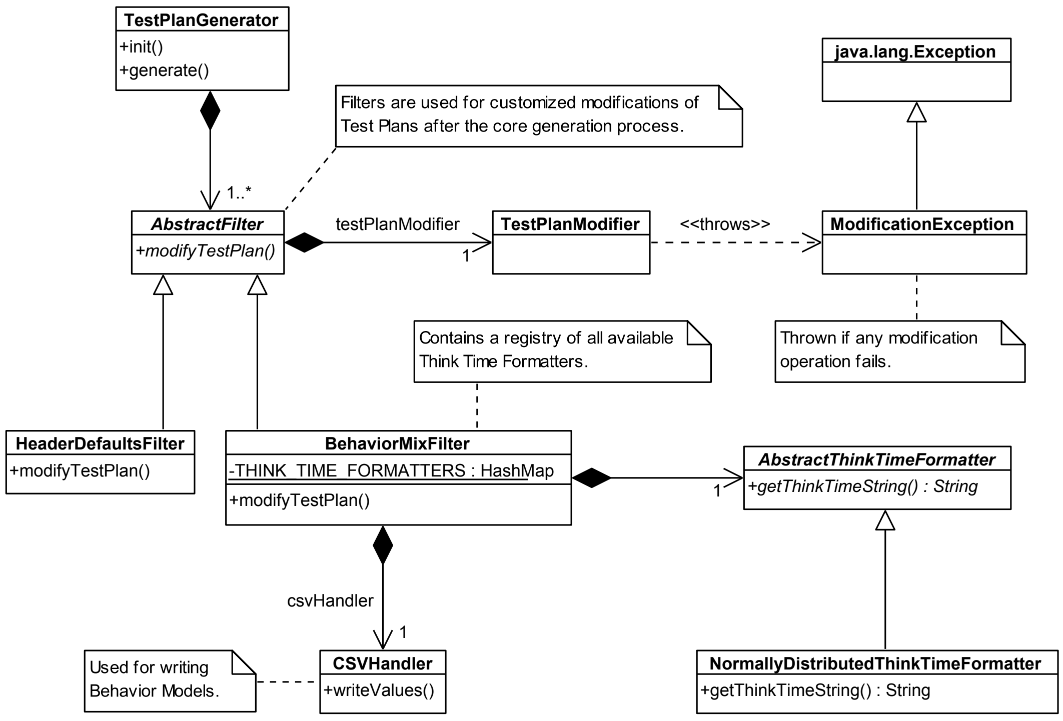

TestPlanGenerator
+init()
+generate()
java.lang.Exception
Filters are used for customized modifications of Test Plans after the core generation process.
1..*
AbstractFilter
+modifyTestPlan()
testPlanModifier
1
TestPlanModifier
<<throws>>
ModificationException
Contains a registry of all available Think Time Formatters.
Thrown if any modification operation fails.
HeaderDefaultsFilter
+modifyTestPlan()
BehaviorMixFilter
-THINK_TIME_FORMATTERS : HashMap
+modifyTestPlan()
1
AbstractThinkTimeFormatter
+getThinkTimeString() : String
csvHandler
1
Used for writing Behavior Models.
CSVHandler
+writeValues()
NormallyDistributedThinkTimeFormatter
+getThinkTimeString() : String

Figure 3.15. Class model of the Test Plan Generator, concerning applicable filters.

▷ *BehaviorMixFilter* installs the Behavior Mix into a Test Plan by writing the workload intensity and the relative frequencies of the Behavior Models into the Markov Session Controller. Furthermore, Behavior Models stored in the M4J-DSL model are written to CSV files. Therefore, the filter requires an instance of *CSVHandler*.

▷ *TestPlanModifier* provides helping methods for modifying any Test Plans, including the search, insertion, removal, or replacement of elements. If any modification error occurs, a *ModificationException* will be thrown.

▷ *AbstractThinkTimeFormatter* is the abstract base class for any think time formatters. It includes a *getThinkTimeString()* method which returns an appropriate string representation of the regarding think time type, for being stored in Behavior Model matrices.

▷ *NormallyDistributedThinkTimeFormatter* is a formatter for think times which follow a normal distribution, with each think time being defined by the use of a mean and a standard deviation value. Further details regarding to think times will be discussed in Section 5.2.

External Libraries

The Test Plan Generator requires a large set of external libraries, mainly caused by the OCLinEcore validation framework. Most libraries have been taken from the Eclipse Kepler (Eclipse 4.3) release and should be updated commonly for any newer Eclipse version. Even for the JMeter API, including the Markov4JMeter add-on, several libraries have been included. These libraries need to be updated in terms of the currently used version of JMeter and Markov4JMeter respectively, to ensure that generated Test Plans comply to the currently used XMI format. A complete table of external libraries is given in Table C of Appendix C. Those libraries are mostly running under the Eclipse Public License (EPL) and Apache License (AL) 2.0 respectively.

3.2.4 Extensibility

This section shortly describes how the Test Plan Generation Framework can be extended with respect to additional Test Plan elements and functionality. It illustrates the definition of new command-line options, Test Plan elements and structures, Think Time Formatters, Request Transformers, and filters. Implementation details should be obtained from the corresponding source code.

▷ Any new command-line options should be registered in the *CommandLineArgumentsHandler* class. This class manages all available options including their short and long names. Those options are read by the *main()* method, which should be extended accordingly.

▷ New Test Plan elements to be supported by the Test Plan Factory must be implemented in its corresponding class. The implementation is straight forward, and the class offers helping methods as well as code examples from which the new code can be easily derived.

▻ Alternative Test Plan structures can be implemented by creating a subclass of *AbstractTestPlanTransformer*. Such subclass must define the newly supported structure in the *transform()* method. A more comfortable way would be the use of a DSL which will be discussed as an open issue in Section 3.2.5.

▻ Think Time Formatters to be added to the framework must inherit from class *AbstractThinkTimeFormatter* and be registered in class *BehaviorMixFilter*, for being applicable on related *ThinkTime* instances of a given M4J-DSL model.

▻ Analogous to Think Time Formatters, Request Transformers must inherit from class *AbstractRequestTransformer* and be registered in *AbstractProtocolLayerEFSMTransformer*.

▻ Filters to be added to the framework must inherit from class *AbstractFilter*. The integration of new filters is still an open issue which will be discussed in Section 3.2.5.

3.2.5 Limitations and Future Work

Since the implementation focus has been put on the framework's core functionality, several open issues and possible improvements of the Test Plan Generator have been identified during its development process. These points aim to the extension of the framework's functionality and its usability as well. They might constitute a basis for future development.

▻ *GOTO support for Test Plans*. As already discussed in Section 3.2.3, JMeter Logic Controllers for GOTO statements do not exist, which complicates the modeling of Protocol Layer EFSMs. As a solution, two approaches might be pursued: implementing an own *GOTO Controller* for JMeter or, if not possible, using the equivalence of *GOTO-* and *WHILE-programs* [Ashcroft and Manna, 1979]; *IF/WHILE Controllers* are available in JMeter. Since a Protocol Layer EFSM can be interpreted as a GOTO-program, with each state denoting a statement and each transition (including its possible guard) denoting an (IF-)GOTO statement, a GOTO program can be built up from an EFSM. Such program can be converted to a WHILE program which only uses IF- and WHILE-statements. A Test Plan can be built accordingly. One drawback of this approach is the complexity of the resulting structure which would be hard to maintain. However, *IF-* and *WHILE-Controllers* are already supported by the Test Plan Generator.

▻ *Grammar for defining Test Plan structures*. The core structure of a generated Test Plan currently depends on a generic class which builds Test Plans for any supported request type. Therewith, each resulting Test Plan contains elements which might be unnecessary for certain request types. For example, there is no point in using an *HTTP Cookie Handler* for a Test Plan which only sends Java requests. Even in context of minor structural changes, it turned out that a grammar for defining the core structure of generated Test

Plans would be of great help instead of implementing various structure classes. *Xtext* [The Eclipse Foundation, 2014c] supports powerful tools for defining such a grammar including a dedicated editor.

- *UI front-end*. Passing the input parameters via command-line to the Test Plan Generator is sufficient for quick configuration purposes, but the amount of parameters is in many cases not easy to be handled that way. For a more comfortable usage of input parameters, a lightweight – possibly text-based – UI front-end using neither much memory nor much central processing unit (CPU) power would be of great help.
- *Advanced support of filters*. Consistent filter handling is still unimplemented, that is filter options cannot be passed via command-line yet. In particular, an optimization filter for storing common values globally, as described in the *Filters* paragraph of Section 3.2.3 would be of great help.
- *Immediate start of tests*. The Test Plan Generator supports an immediate start of the JMeter core engine, for running tests without starting the JMeter application explicitly. For this feature, test results need to be captured and processed suitably, which still is work in progress.

Chapter 4

Markov4JMeter Workload Model Creation

This chapter discusses the *Markov4JMeter Workload Model Creation* part of the implementation approach, as illustrated in the lower left corner of Figure 1.1. An explicit overview of the creation process, illustrating the assembly status of the included parts, is given in Figure 4.1. Green check marks denote transformation input/output pipes which have been successfully implemented. The orange check mark including the works sign indicates a partially completed implementation with open issues which remain unsolved so far. The marked parts will be covered in the following discussion. Section 4.1 presents a framework for transforming Screenflow information of a SUT into M4J-DSL models. Section 4.2 introduces a Java Sampler which builds on Java Reflection techniques, to be used for the Protocol Layer of generated M4J-DSL models.

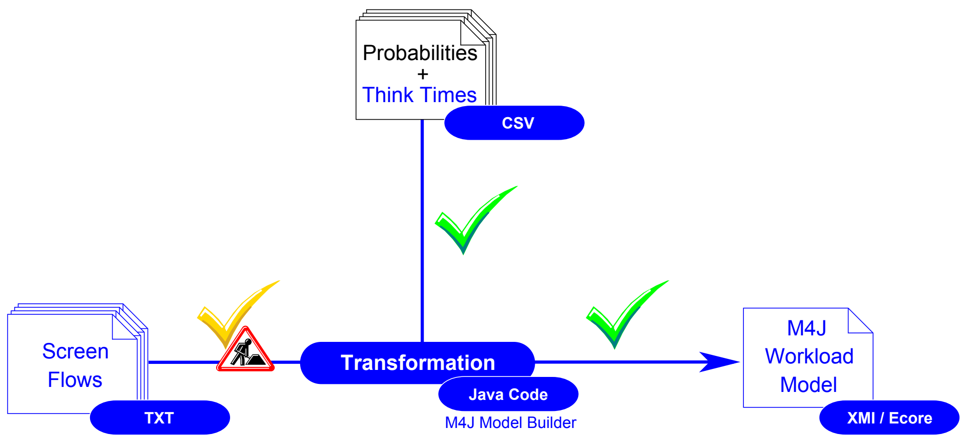


Figure 4.1. Overview of the M4J-DSL model creation process, illustrating the assembly status of the included parts.

4.1 M4J-DSL Model Generation Framework

The framework for creating Markov4JMeter Workload Models, shortly referred to as *M4J-DSL Model Generator*, builds for a given set of Screenflows an M4J-DSL model, which can be passed as input to the Test Plan Generator introduced in Section 3.2. In the Behavior Models of the resulting M4J-DSL model, probabilities and think times which originate from monitoring data, will be installed additionally. Currently, the framework is adapted to b+m gear Screenflows, but its architecture can be modified in certain ways, for being used in combination with other systems, which provide similar Screenflow information. The M4J-DSL Model Generator has been implemented as a standalone application in Java and is executable from command-line.

Section 4.1.1 gives a brief overview of the essential process performed by the generator when it transforms a set of input flow data, such as b+m gear Screenflows, to an M4J-DSL model. Section 4.1.2 describes the usage of the M4J-DSL Model Generator, including a more detailed introduction to its features. Section 4.1.3 outlines the architecture of the framework, illustrating its package structure and class model. Section 4.1.4 describes the extensibility of the framework in terms of its core functionality and possible new features. Section 4.1.5 conclusively discusses limitations and open issues for future work.

4.1.1 Overview

The primary task of the M4J-DSL Model Generator is the transformation of Screenflows and user behavior information to a Markov4JMeter workload model, as illustrated in Figure 4.1. In addition to the resulting M4J-DSL model, the framework generates a graph representation of the Session Layer EFSM optionally, for facilitating the analysis of any EFSM's structure. The transformation process is based on a set of configuration properties. The input Screenflows need to be provided as text files, which originate from the generation process of the targeted application. Hence, these generally files do not have a standard format. Probabilities and think times are given as CSV files, including (partial) Behavior Models which have been extracted from monitoring data, as discussed later in Section 5.1. The resulting M4J-DSL model is stored as an (XMI-serialized) Ecore object. Graphs are stored as DOT-formatted files, which can be displayed or converted to alternative image formats through the graph visualization software *Graphviz* [The Eclipse Foundation, 2014d]. Additional visualizations, e.g., for Protocol Layer EFSMs, might be added, which constitutes a future work issue.

Figure 4.2 shows the internal input-output transformation process of the M4J-DSL Model Generator. In the *Generator Initialization* task, several configuration steps are included, e.g., parsing the command-line parameters, reading the configuration properties, and organizing the output file paths. After these setup tasks, the input Screenflows are parsed. If parsing fails, the generator does not produce any output files and gives an error message instead, indicating the parsing error which occurred. In case Screenflow parsing is successful, the *Model Transformation* task is started. An M4J-DSL model corresponding

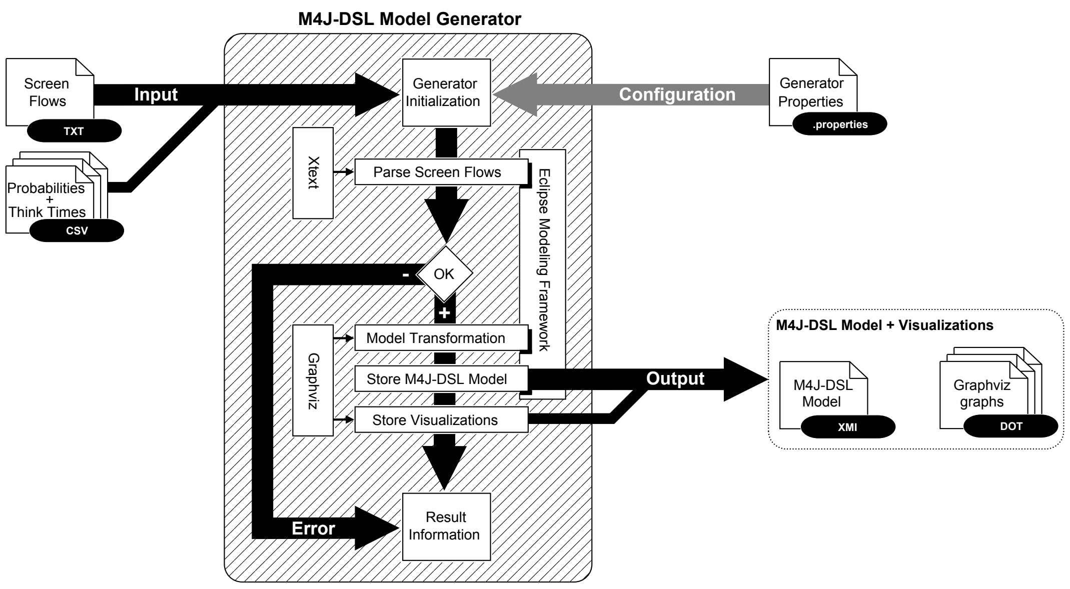


Figure 4.2. Input-output transformation process of the M4J-DSL Model Generator.

to the structure indicated by the Screenflows is generated and enriched with probabilities and think times provided by the regarding input information. It is important to note that this information does not necessarily cover the probabilities and think times between *all* Markov states that result from Screenflows; it might cover a small subset only, depending on the monitoring which has been applied on the targeted SUT. After the transformation, the M4J-DSL model is written to an XMI file, and graphs, which have been generated optionally besides that model, are stored as DOT files, for being displayed by Graphviz. Finally, the generator gives a result information, indicating possible failure or success.

The parsing of Screenflows is based on the Xtext framework which has been introduced in Section 2.4.2. Furthermore, the underlying core technology for the parser as well as its two subsequent tasks is EMF. Visualization graphs that are generated (secondarily) in the *Model Transformation* respectively *Store Visualizations* tasks must follow a DOT format which complies to the Graphviz specification.

4.1.2 Usage

The M4J-DSL Model Generator is implemented as a standalone application in a single JAR file, for being executed from command-line or included as external library into other Java projects. It creates an M4J-DSL model, based on a set of input Screenflows which need to be provided as textfiles in a single folder of the file system. In the same way, CSV-formatted probabilities and think time information must be passed as input to the generator. The resulting model can be used as input for the Test Plan Generator which has been introduced in Section 3.2, or it can be loaded into an EMF Form Editor for being modified. An excerpt of b+m gear-specific Screenflows which have been extracted from the CarShare application will be discussed next, illustrating their transformation to a Session Layer EFSM by example.

Configuration

The configuration of the M4J-DSL Model Generator is based on a property file which must be passed to the application via command-line parameter. The configuration also supports the definition of workload intensity. Therefore, it provides properties for defining the workload intensity type as well as a formula string. Figure 4.3 shows an example of a workload intensity definition. By now, *constant* is the only supported type, assuming that a fixed number of users is emulated during a JMeter experiment, possibly slightly varying through ramp-up times in a JMeter experiment. This limitation can be solved, which constitutes an open issue discussed in Section 4.1.5.

Command-Line Parameters

Parameters passed via command-line to the M4J-DSL Model Generator must be specified in a certain order. Table 4.1 gives an overview of these parameters; examples will be given below.

Example Parameters

▻ The parameter sequence

```
generator.properties ./flows/ ./behavior/ workloadmodel.xmi
```

denotes a minimum start configuration for the M4J-DSL Model Generator, using configuration values provided by file "generator.properties". Input Screenflows and behavior information files are read from folders "./flows/" and "./behavior/" respectively. The output file for the resulting M4J-DSL model is "workloadmodel.xmi"; no graph visualization files will be written. Paths and filenames might be put into quotes optionally.

```
workloadIntensity.type    = constant
workloadIntensity.formula = 1
```

Figure 4.3. Example of a workload intensity definition in the configuration file of the M4J-DSL Model Generator (excerpt); the workload intensity is set to a constant number of 1 user(s).

Table 4.1. M4J-DSL Model Generator command-line parameters. *Index* denotes the position of a parameter in a sequence being passed via command-line.

Index	Parameter Description
0	Properties file which provides the configuration values of the M4J-DSL Model Generator, e.g., "generator.properties".
1	Path to an existing directory which contains the input Screenflow information to be transformed, e.g., "./examples/flows/".
2	Path to an existing directory which contains the user behavior information to be installed in the resulting M4J-DSL model, e.g., "./examples/behavior/".
3	XMI output file of the resulting M4J-DSL model, e.g., "WorkloadModel.xmi".
	Optional Arguments
4	(Optional) output file of the visualization graph, e.g., "./output/graph.dot" (should be a directory for multiple files in future versions); if not defined, no graph will be generated.

▻ The parameter sequence

```
generator.properties ./flows/ ./behavior/ workloadmodel.xmi graph.dot
```

has the same effect as the first one, but it additionally defines an output file for the visualization graph, to be located in the current directory.

b+m gear Screenflows

The definition of b+m gear Screenflows is based on diagrams with nodes and transitions; each Screenflow has a unique start node and possibly multiple end nodes [Reimer, 2013]. Transitions between nodes might be labeled with events, guards, and actions. A transition fires, if its assigned event occurs and its guard is “true”; in that case, the action is applied initially, when the transition fires [Reimer, 2013]. Figure 4.4 shows an example of a b+m gear Screenflow diagram, illustrating a synchronous call within the *CarShare* application that

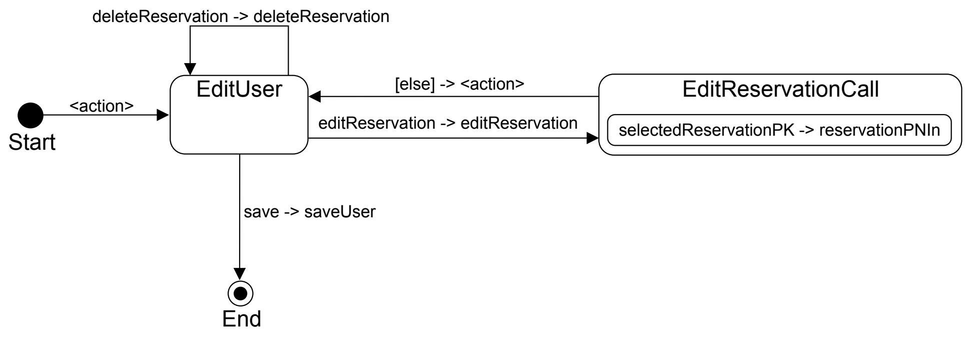

deleteReservation -> deleteReservation
EditUser
[else] -> <action>
EditReservationCall
<action>
Start
editReservation -> editReservation
selectedReservationPK -> reservationPNIn
save -> saveUser
End

Figure 4.4. Screenflow diagram illustrating a synchronous call, based on Reimer [2013, p.55].

has been introduced in Section 2.5.3. It depicts an *EditUser* node representing a View. That View is entered on start of the modeled Screenflow, indicated by an according transition. Reimer [2013] describes the flow as follows: the *EditUser* View allows input of user data; if a *save* event occurs, that data will be stored through a *saveUser* action; analogous, a *Reservation* entity might be deleted. The action which is assigned to an outgoing View transition is implemented in the underlying ViewController. In case an *editReservation* event occurs, another Screenflow named *EditReservation*, which itself allows the modification of reservation data, is called synchronously; the key variable *selectedReservationPK* specifies the item to be modified. As indicated in the diagram, the *EditReservation* Screenflow returns the control to the caller afterward.

Most of the depicted Screenflow information can be extracted from the (*Xtend2*-based) generation process of a b+m gear application. Figure 4.5 depicts two example Screenflows

```
Flow EditUser {
Node Start {
    Transition { event="", guard="", action="", target=EditUser } }
Node EditUser {
    Transition { event="deleteReservation", guard="", action="deleteReservation", target=EditUser }
    Transition { event="reservationChanged", guard="", action="reservationChanged", target=EditUser }
    Transition { event="editReservation", guard="", action="editReservation", target=EditReservationCall }
    Transition { event="save", guard="", action="saveUser", target=End }
    Transition { event="", guard="End", action="", target=EditUser }
    Transition { event="", guard="else", action="", target=EditUser } }
Node EditReservationCall { }
Node End { }
}
```

(a) Screenflow*EditUser*

```
Flow EditReservation {
Node Start {
    Transition { event="", guard="", action="", target=ReservationLocation } }
Node ReservationLocation {
    Transition { event="search", guard="", action="search", target=ReservationLocation }
    Transition { event="locationChanged", guard="", action="locationChanged", target=ReservationLocation }
    Transition { event="selectCar", guard="", action="selectCar", target=ReservationCar }
    Transition { event="exit", guard="", action="", target=End } }
Node ReservationCar {
    Transition { event="back", guard="", action="back", target=ReservationLocation }
    Transition { event="save", guard="", action="saveReservation", target=ReservationSuccess } }
Node ReservationSuccess {
    Transition { event="close", guard="", action="", target=End } }
Node End { }
}
```

(b) Screenflow*EditReservation*

Figure 4.5. Example Screenflows for the CarShare application

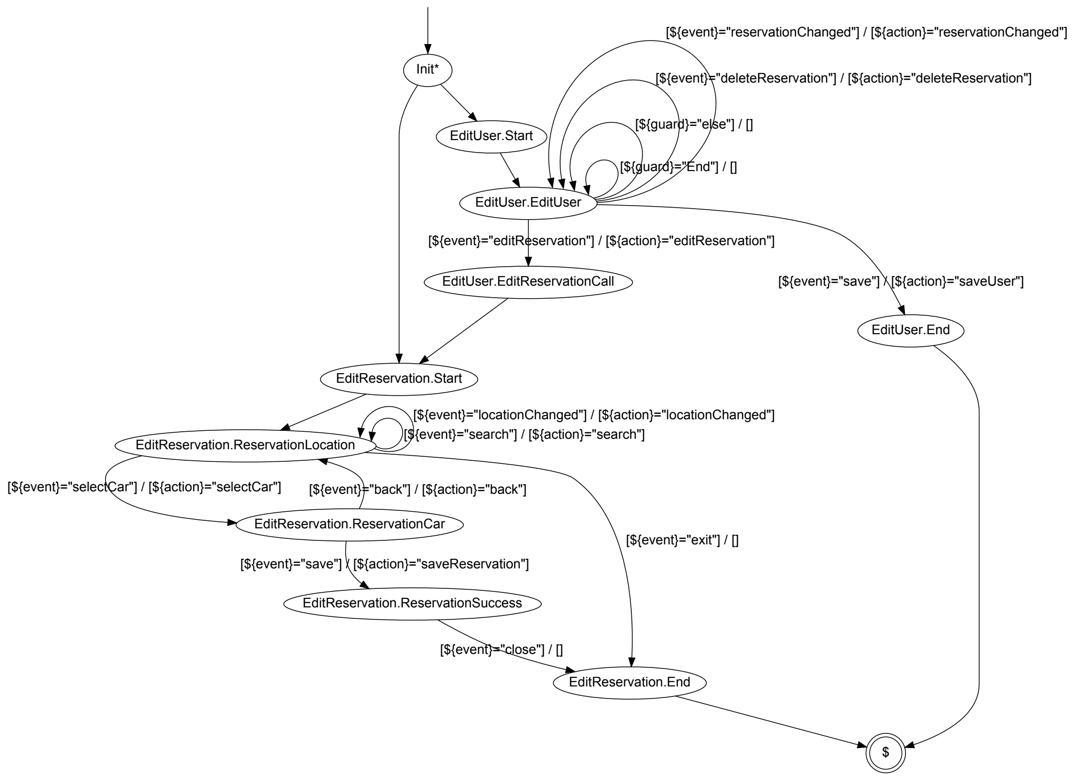

[${event}="reservationChanged"] / [${action}="reservationChanged"]
Init*
[${event}="deleteReservation"] / [${action}="deleteReservation"]
[${guard}="else"] / []
EditUser.Start
[${guard}="End"] / []
EditUser.EditUser
[${event}="editReservation"] / [${action}="editReservation"]
EditUser.EditReservationCall
[${event}="save"] / [${action}="saveUser"]
EditUser.End
EditReservation.Start
[${event}="locationChanged"] / [${action}="locationChanged"]
[${event}="search"] / [${action}="search"]
EditReservation.ReservationLocation
[${event}="selectCar"] / [${action}="selectCar"]
[${event}="back"] / [${action}="back"]
EditReservation.ReservationCar
[${event}="exit"] / []
[${event}="save"] / [${action}="saveReservation"]
EditReservation.ReservationSuccess
[${event}="close"] / []
EditReservation.End
$

Figure 4.6. Example for a Session Layer EFSM, generated from a set of Screenflows; the underlying Screenflows are those of Figure 4.5.

for the CarShare application which have been extracted during the generation process of that system. They have been taken from a total set of 21 Screenflows. Each Screenflow data set includes a (unique) Screenflow name and a list of nodes. The outgoing transitions of a View are listed under the regarding node. An end node is indicated through an empty transition list; a node without transitions might even represent a Screenflow call, if a Screenflow of corresponding name – possibly without *Call* suffix – exists; an example is given by the *EditReservationCall* node in Figure 4.5a.

Transformation Process Screenflows as depicted in Figure 4.5 indicate the Session Layer EFSM of an application-related Markov4JMeter model. The transformation process of the M4J-DSL Model Generator constructs that EFSM for a corresponding M4J-DSL model. Figure 4.6 shows an example of a resulting Session Layer EFSM. Screenflow nodes are mapped to Application States, that is, the Session Layer EFSM is View-based, with each state representing a b+m gear View; state names are fully qualified, using a $Screenflow.View$ notation. Transitions are transformed accordingly, whereas the transformation is not straight forward, since the mapping between b+m gear Screenflows and a Session Layer EFSM of a Markov4JMeter workload model does not match perfectly. The following mismatches have been identified:

▷ *Different format of transition labels.* The transition labels of b+m gear and Markov4JMeter differ in the following two points:

- Guards in b+m gear denote identifiers, in contrast to conditional expressions as in Markov4JMeter.

  As a solution, those identifiers are associated with a JMeter $\$\{guard\}$ variable in the resulting M4J-DSL model. That variable needs to be set accordingly during the runtime of a JMeter experiment by the used Samplers. The guard can be evaluated in terms of the variable value.

- The Markov4JMeter add-on currently does not support the modeling of events. On the other side, the b+m gear event information is not redundant, since transitions are distinguished by events. For example, the View *EditUser* of the same-named Screenflow in Figure 4.5a contains four transitions with event definitions, but undefined guards. In case the events are ignored, transitions would coincide, which would lead to clashing target states. Consequently, a flow controller would not know which transition to fire.

  As a solution, the events are modeled as guards, possibly combined with existing guards by logical AND. In addition, events are associated with an $\$\{event\}$ variable, which needs to be set during runtime, as discussed for the guard variable before

▷ *Multiple initial and exit states.* Screenflows must not necessarily have interrelations, which possibly leads to multiple initial states for a given set of Screenflows, since each start node of a Screenflow indicates an individual initial state. Furthermore, Screenflows might have several End nodes, implying multiple exit states.

As a solution, additional (unique) initial and exit states are created, to be linked to any state which represents the start respectively end node of a Screenflow. However, this is not sufficient, since certain Views in that way might become active, though they are not allowed otherwise. Figure 4.5a, includes such a situation, as the View *EditReservation.Start* might become active, though no user has been edited before. This is still an open issue, to be further discussed in Section 4.1.5.

The M4J-DSL Model Generator does not build application-related Protocol Layer EFSMs, since appropriate information cannot be obtained from the generation process of a b+m gear application currently. Therewith, the resulting M4J-DSL model uses default Protocol Layer EFSMs, to pass the validation process, performed by the Test Plan Generator, successfully. The integration of Protocol Layer related information will be discussed as a future work later.

Result Output The generator output consists of a M4J-DSL model and Graphviz files, as indicated by Figure 4.2. These files will be written as XMI file respectively DOT files into the output file path that has been specified via command-line parameter. Currently, only the Session Layer EFSM is written as Graphviz graph, but graphs for Protocol Layer EFSMs might follow.

4.1.3 Architecture

This section gives an overview of the M4J-DSL Model Generator's architecture, starting with an introduction of the package structure. Afterward, an outline of the class structure will be given, discussing the main classes in more details. The section closes with an overview of external libraries used for the framework. Any code details which are not covered by the description should be obtainable from the related source code.

Package Structure

The packages of the M4J-DSL Model Generator are organized in a mainly hierarchical order. An overview of the structure is given in Figure 4.7, which depicts the nested packages of the *net.sf.markov4jmeter.m4jdslmodelgenerator* root package. The content of the individual

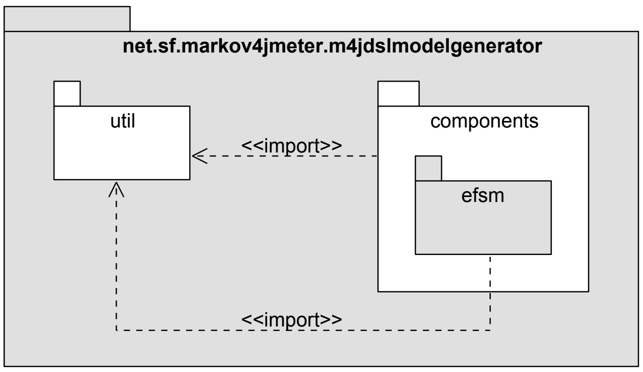


Figure 4.7. Package structure of the M4J-DSL Model Generator

(sub-)packages is as follows:

▷ The *util* package provides, analogous to the *util* package of the Test Plan Generator, general-purpose classes for simplifying miscellaneous base tasks. These classes are intended for graph visualization, ID generation, (de-)serialization of Ecore objects, and CSV file reading. Most of them are not framework-specific and can be simply integrated into alternative systems.

▻ The *components* package contains the generation classes for each component of the Markov4JMeter workload model, that is, classes for the Application Model, Behavior Mix, Behavior Models, and workload intensity respectively.

▻ The *components.efsm* package includes classes for building the Protocol Layer EFSMs and Session Layer EFSMs of the output model.

Class Structure

The class model of the M4J-DSL Model Generator consists of 17 classes. An overview is given in Figure 4.8, whereas attributes and operations are not intended to be complete; they serve for illustration purposes only. As for previously discussed class diagrams, the signatures of operations must not necessarily comply to the signatures in the source code, to avoid confusion caused by large parameter lists.

▻ *M4jdslModelGenerator* is the base class with the `main()` method for standalone execution. It particularly contains the code for the main logic, as well as code for reading command-line parameters and application properties.

▻ *ApplicationModelGenerator*, *BehaviorMixGenerator*, *BehaviorModelsGenerator*, and *WorkloadIntensityGenerator* build the M4J-DSL model parts which represent the Application Model, Behavior Mix, Behavior Models, and workload intensity respectively.

▻ *IdGenerator* allows the generation of identifiers for certain M4J-DSL model elements.

▻ *AbstractProtocolLayerEFSMGenerator* is the abstract base class for any generator class which is intended to build the Protocol Layer EFSMs of an M4J-DSL model.

▻ *GearProtocolLayerEFSMGenerator* is a subclass of *AbstractProtocolLayerEFSMGenerator* for building b+m gear-specific Protocol Layer EFSMs.

▻ *AbstractSessionLayerEFSMGenerator* is the abstract base class for any generator class which is intended to build the Session Layer EFSM of an M4J-DSL model. It additionally provides helper methods for building M4J-DSL model elements of certain type, such as services and Application States.

▻ *FlowSessionLayerEFSMGenerator* is a subclass of *AbstractSessionLayerEFSMGenerator* for building Session Layer EFSMs, based on b+m gear Screenflows.

▻ *FlowDSLParser* supports the reading of b+m gear Screenflow files. It builds on Xtext and allows therewith a simple adaption of alternative input flow formats, since a corresponding parser can be easily obtained through appropriate modification of the underlying Xtext grammar. This will be further discussed in Section 4.1.4.

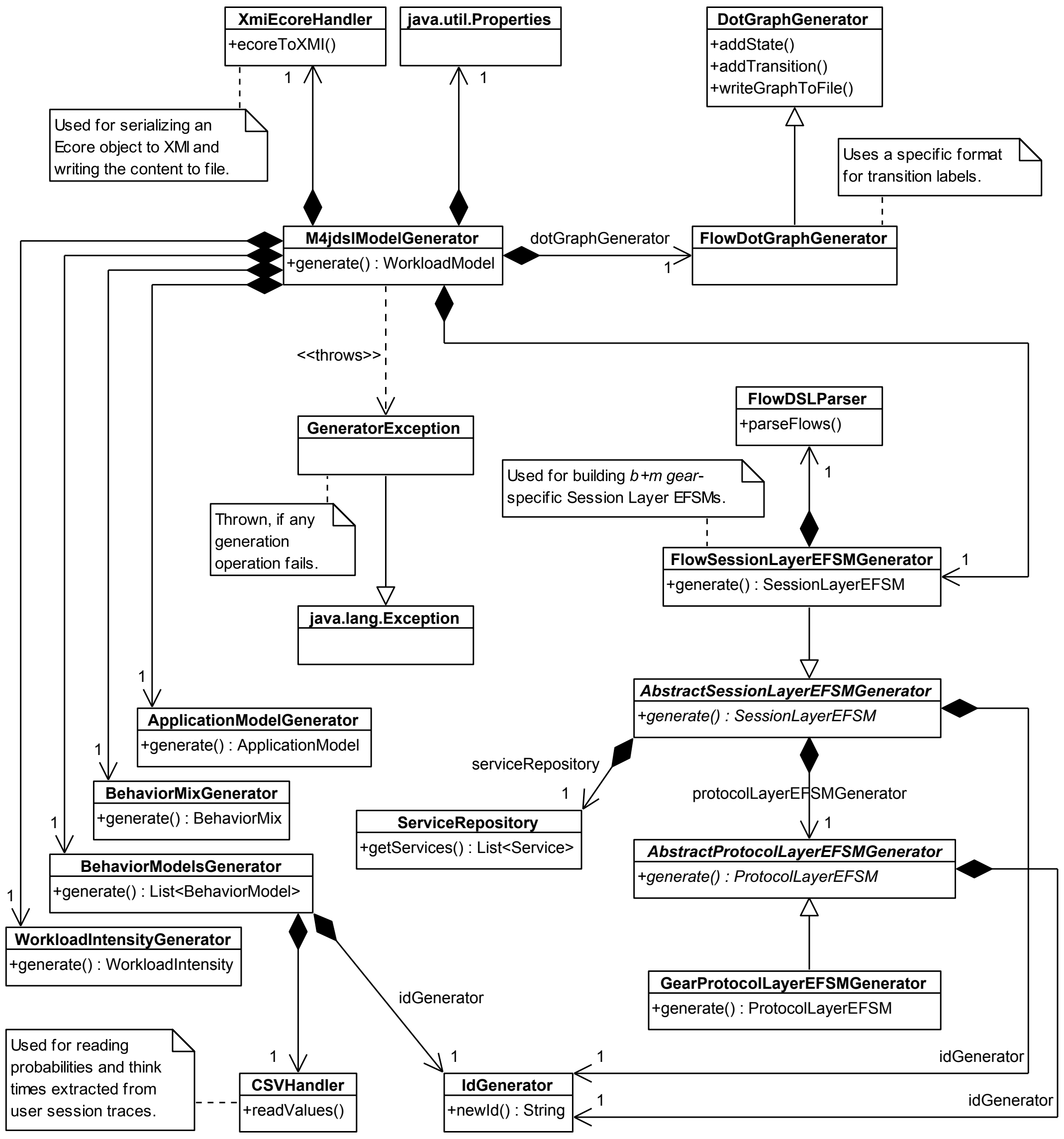

XmiEcoreHandler
+ecoreToXMI()
java.util.Properties
DotGraphGenerator
+addState()
+addTransition()
+writeGraphToFile()
Used for serializing an Ecore object to XMI and writing the content to file.
Uses a specific format for transition labels.
M4jdslModelGenerator
+generate() : WorkloadModel
dotGraphGenerator
FlowDotGraphGenerator
<<throws>>
GeneratorException
Thrown, if any generation operation fails.
java.lang.Exception
FlowDSLParser
+parseFlows()
Used for building b+m gear-specific Session Layer EFSMs.
FlowSessionLayerEFSMGenerator
+generate() : SessionLayerEFSM
AbstractSessionLayerEFSMGenerator
+generate() : SessionLayerEFSM
serviceRepository
protocolLayerEFSMGenerator
ApplicationModelGenerator
+generate() : ApplicationModel
BehaviorMixGenerator
+generate() : BehaviorMix
ServiceRepository
+getServices() : List<Service>
BehaviorModelsGenerator
+generate() : List<BehaviorModel>
AbstractProtocolLayerEFSMGenerator
+generate() : ProtocolLayerEFSM
WorkloadIntensityGenerator
+generate() : WorkloadIntensity
GearProtocolLayerEFSMGenerator
+generate() : ProtocolLayerEFSM
idGenerator
Used for reading probabilities and think times extracted from user session traces.
CSVHandler
+readValues()
IdGenerator
+newId() : String
idGenerator
idGenerator
1

Figure 4.8. Class model of the M4J-DSL Model Generator; method return types in `Generator` classes denote (lists of) M4J-DSL types.

▻ *ServiceRepository* serves for collecting all services during the generation process. The class provides methods for registering, requesting, and finding services by name. It additionally ensures that services are *unique*, that is, multiple instances representing the same service are not allowed to exist.

▻ *DotGraphGenerator* simplifies the creation of graph representations for any Session Layer EFSM. Such graph representations will be stored in DOT-formatted files, which can be loaded and visualized by the Graphviz application. The class particularly supports the registration of new states and transitions for being written appropriately formatted into a DOT file.

▻ *FlowDotGraphGenerator* is a subclass of *DotGraphGenerator* for using specific transition labels, formatted in `[`*guard*`]/[`*action*`]` notation. It provides corresponding transition registration methods, expecting guards and actions as parameters for accordingly formatted labels.

▻ *XmiEcoreHandler* encapsulates the code for the (de-)serialization of Ecore models. The M4J-DSL Model Generator uses it for writing an output M4J-DSL model to an XMI file.

▻ *CSVHandler* provides methods for reading (and writing) CSV files, e.g., files which supply probabilities and think times extracted from session traces.

External Libraries

The amount of external libraries required for the M4J-DSL Model Generator is mainly caused by the Xtext parsing unit. Additional libraries are required for the Ecore modeling utilities, including the (de-)serialization of corresponding models. All of those libraries originate from the Eclipse Kepler (Eclipse 4.3) release and should be updated commonly for any newer Eclipse version, to keep the compatibility between them. A complete table of external libraries, which mostly run under the EPL and AL 2.0 respectively, is given in Table C of Appendix C.

4.1.4 Extensibility

The M4J-DSL Model Generator has been developed with regard to flexibility and extensibility. It currently supports b+m gear Screenflows, but alternative flow formats can be adapted easily. Since the parsing unit for the flows is Xtext-based, alternative parser code can be generated in a simple and efficient way. Xtext builds on grammars which describe the formats to be parsed, as shown by example in Listing 4.1. Grammar definitions are located in own Eclipse projects, which allows the modular development of related parsers. Any new parser unit must be included as library to the M4J-DSL Model Generator. Besides a parser, Xtext additionally generates a syntax-highlighted editor for each grammar; therewith, a comfortable testing suite is already provided for any new flow format. However, flow formats generally build on individual information and structures. Consequently, the EFSMs for Session Layer and Protocol Layer do not have a common structure. This implies that appropriate subclasses of *AbstractSessionLayerEFSMGenerator* respectively *AbstractProtocolLayerEFSMGenerator* must be implemented, for processing the provided information according to the considered structure. For the same reason, a dedicated *DOTGraphGenerator* class with minor adjustments to its output format might be needed.

```
grammar net.sf.markov4jmeter.gear.FlowDSL with org.eclipse.xtext.common.Terminals

generate flowdsl "http://flowdsl/1.0"

FlowRepository:
    (flows += Flow)*;

Flow:
    'Flow' name=ID '{'
        (nodes += Node)*
    '}';

Node:
    'Node' name=ID '{'
        (transitions += Transition)*
    '}';

Transition:
    'Transition' '{' event=Event ',' guard=Guard ',' action=Action ','
        target=Target '}';

Event:
    'event' '=' value=STRING;

Guard:
    'guard' '=' value=STRING;

Action:
    'action' '=' value=STRING;

Target:
    'target' '=' value=ID;
```

Listing 4.1. Example for an Xtext grammar, specifying the b+m gear Screenflows format.

4.1.5 Limitations and Future Work

The open issues and possible improvements of the M4J-DSL Model Generator which have been identified during its development process, mainly aim to the usability and flexibility of the application. The following points might constitute a basis for future development.

▷ *Mapping between b+m gear and Markov4JMeter models.* As illustrated in Section 4.1.2, several mismatches in context of the mapping between both considered models have been identified. A straight M2M transformation remains difficult, since certain solution steps imply much additional implementation work. This will be further discussed in Section 6.1.2.

▷ *Protocol Layer.* As discussed in Section 4.1.2, the generation of the Protocol Layer EFSMs is not supported by the framework yet, since appropriate information is currently not available from the generation process of b+m gear applications. This denotes an open issue which will be further discussed in Section 6.1.2.

▷ *UI front-end.* A more comfortable and flexible handling of command-line parameters is still an open implementation issue; parameter flags should be added. A lightweight UI fron-end, as already discussed in context of the Test Plan Generator, would be a preferable solution.

▷ *Advanced graph visualization.* It turned out that the visualization of EFSMs denotes a very helpful utility for analyzing the result structures. Hence, an implementation even for Protocol Layer EFSMs is recommended. For oversized graphs, additional display features, such as colored transitions, might facilitate the identification of certain graph elements.

▷ *Workload intensity.* As discussed previously, the current implementation supports a constant workload intensity only. For various types, the M4J-DSL needs to be extended accordingly, and new types need to be registered in the M4J-DSL Model Generator.

4.2 Reflective Java Sampler Client

JMeter provides several Sampler types for invoking Java methods from within a Test Plan, as discussed in Section 2.7. A *BeanShell Sampler* constitutes a suitable solution for executing service methods of a targeted application, by the use of appropriate Java statements. However, this Sampler type includes a Java source interpreter, which works possibly too slow to gain representative results in a (locally) conducted performance test. This section presents an approach to overcome this drawback and introduces therefore a JMeter Java Sampler Client, which builds on Java Reflection techniques.

As discussed in Section 2.7, JMeter provides a *Java Sampler*, which can be used to invoke methods of corresponding client classes. Therefore, the code of such clients must be implemented by the tester. This is contrary to the aim of Test Plan generation, since specific code, e.g., a service method invocation in the regarding test application, must be implemented manually in a client class; it generally requires enormous effort to cover all test cases. The idea for the *Reflective Java Sampler Client* is to implement *generic* structures in the client class, by the use of the Java Reflection package. Java Samplers which use the client for calling a Java method, need to provide the method signature and appropriate parameters.

Since the client constitutes an enhancement rather than being fundamental for the Test Plan generation approach, its architecture will not be discussed in details. Required information should be obtainable from the code itself. Section 4.2.1 gives an overview of the client. Section 4.2.2 shortly explains how the client can be used in JMeter. Section 4.2.3 illustrates limitations and future work.

4.2.1 Overview

Figure 4.9 gives an overview of the Reflective Java Sampler Client. The left side represents the JMeter application, and the right side shows the client. Java requests are sent from a Test Plan to the client. A class or object needs to be specified for calling a static or instance method respectively. Additionally, the method signature must be provided as well as the parameters required for a method call. Optionally, a variable for the return value might be specified. As the Test Plan denotes a text file, objects must be serialized.

The *Sampler Initialization* task of the client retrieves the request information and processes it accordingly. At first, the *Class/Object Lookup* task tries to find the targeted class or object in the test application, followed by the *Method Lookup* task. In the *Arguments Resolution* task,

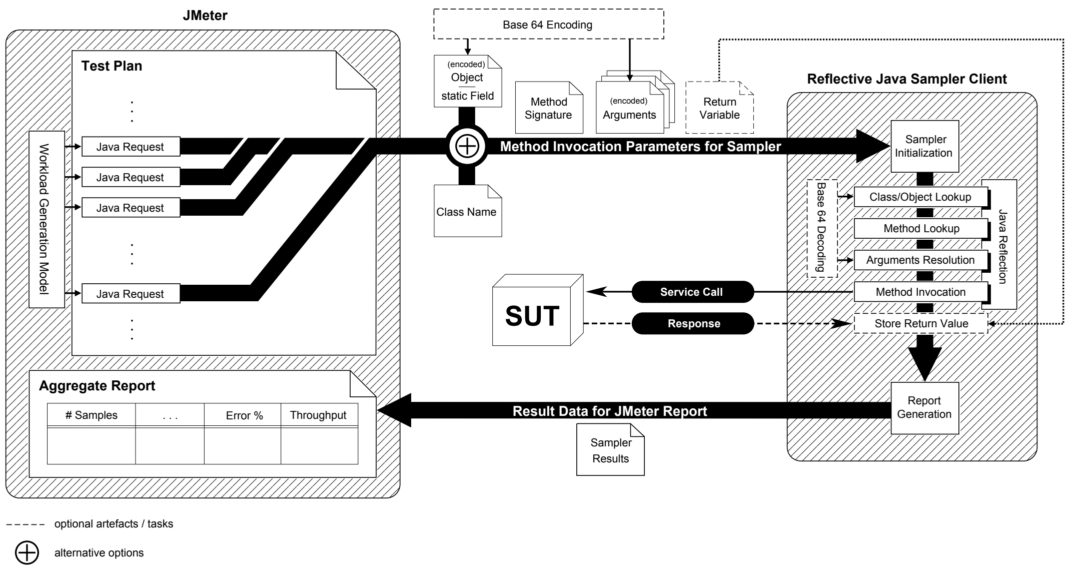


Figure 4.9. Input-output transformation process of the M4J-DSL Model Generator.

serialized objects are reconverted; this is only necessary, if objects originate from a Test Plan. Objects passed from Sampler to Sampler are stored without serialization. Method invocation in the given SUT is done in the *Method Call* task. The *Store Return Value* task puts a possible return value into a corresponding variable. Finally, the client generates a report, which is sent back to the JMeter application, to inform about failure or success of the method call.

4.2.2 Usage

The usage of the client is mainly based on the specification of the JVM [Lindholm et al., 2013]. The installation of Java clients has been already illustrated in Section 2.7. In case a method to be called is static, the (fully qualified) name of its class must be provided, e.g., *java.lang.System* for calling a static method of that class. If a method to be called is an instance method, the regarding object must be provided as a serialized String, whereas the framework builds on *Base 64* encoding. Alternatively, a static field might be specified, e.g., *System.out* for calling the *println()* of the standard output stream object. A method signature is formatted as

$$\mathit{methodName}\texttt{(}\mathit{parameterType}_1\texttt{, }\mathit{parameterType}_2\texttt{, }\ldots\texttt{):}\mathit{returnType}$$

whereas each type must be specified in *field type* notation, as defined in *The Java Virtual Machine Specification - Java SE 7 Edition* [Lindholm et al., 2013, p.78]. For example, the signature

```
calc(I, [[I, [Ljava.lang.String;):F
```

denotes a method named *calc*, with parameters of type `int`, `int[][]` and `String[]`, respectively. The return type is `float`. Field types can be distinguished between *base types*, *arrays* and *object types*.

- ▷ Base types must be specified by their associated JVM characters; Table 4.2 gives an overview.
- ▷ Arrays must be specified as Field Types with leading `[` characters, indicating the regarding array dimension. For example, `[I` denotes an `int[]` array, and `[[F` denotes a `float[][]` array.
- ▷ Object types representing a class or interface, must be specified by their fully qualified names, including a leading `L` character and a closing semicolon. For example, `Ljava.lang.String;` denotes the type `String`.

Parameters must be specified in the configuration panel of each Java Sampler, using the self-explaining input form which becomes active when the Reflective Java Sampler Client is selected by the Sampler.

Table 4.2. Base Types and their associated JVM characters

Base Type	Character
boolean	Z
byte	B
char	C
double	D
float	F
int	I
long	J
short	S

4.2.3 Limitations and Future Work

Limitations are mainly given through the missing evaluation of expressions and complicated usability. Hence, the following points might constitute a basis for further development.

▻ *Evaluation of expressions*. Logical and arithmetic expressions cannot be evaluated by the client yet, since operations for simple data types are not supported. For example, simple increase operations for integer variables, as indicated by `i++`, cannot be computed. A workaround –at least for simple expressions– is given by dedicated methods in a specific class for performing such operations. Those methods can be invoked by the client with proper parameters.

▻ *Complicated usability*. The usability of the Reflective Java Sampler Client is complicated through the high effort, which is required for defining low-level method signatures. A script engine, which transforms standard Java method calls to corresponding low-level calls, would be of great help. The benefit of a working solution would be the saving of much CPU usage, caused by an interpreter during the runtime of an experiment.

Chapter 5

User Behavior Extraction

This chapter covers the extraction of user behavior information from a set of user session traces, which have been monitored on a SUT. The underlying approach, illustrated in Figure 5.1, has been successfully applied in the research project DynaMod [van Hoorn et al., 2011, 2013], and it has been already described by Schulz [2013]. The single steps are approved to be working, which is why green check marks are assigned to all transitions. The remainder of this chapter introduces enhancements only, which are blue-colored in Figure 5.1. Section 5.1 describes the extraction of think times from user session traces. Section 5.2 illustrates the implementations in the Markov4JMeter add-on for emulating the think times accordingly.

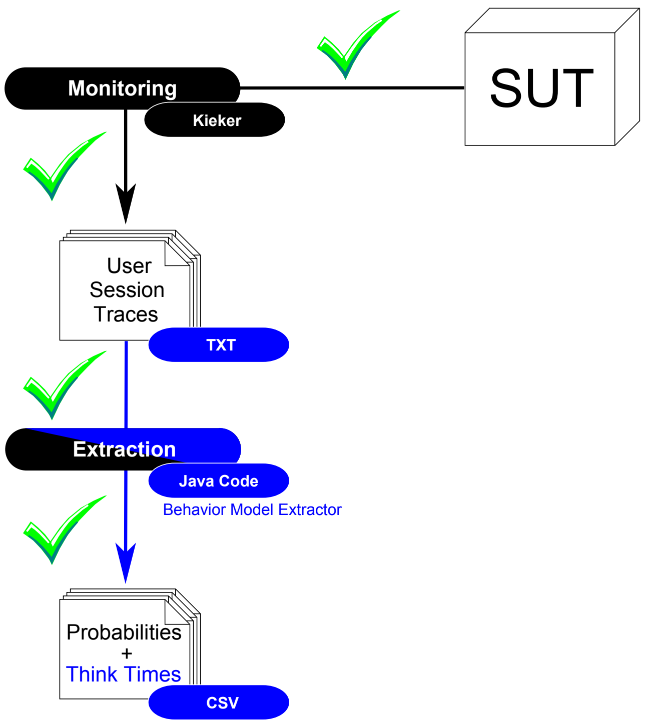


Figure 5.1. Overview of the user behavior extraction process, illustrating the assembly status of the included parts.

5.1 Behavior Model Extraction

For the extraction of Behavior Models from user session traces, a dedicated tool, namely *Behavior Model Extractor* [Schulz, 2013], has been implemented in context of the DynaMod project. This tool supports the extraction of transition probabilities, and think time extraction has been implemented within the scope of this thesis. The tool is now able to calculate think times which follow a normal distribution. Section 5.1.1 outlines the calculation schema.

5.1.1 Normal Distribution of Think Times

The *normal distribution* [Kirkup and Frenkel, 2006] of think times builds on a mean and a standard deviation, which can be calculated from a finite number of delays, extracted from user session traces. Those delays can be retrieved through the time information which is included to monitoring data. If the start and end time of each invoked service is given, the delay between two service calls denotes the difference between the end time of a service and the start time of its successor. After collecting the delays of each outgoing transition for all Markov States (services), the conversion to think times is done as follows: let $D_{ij} = \{d_{ij}^1, \ldots, d_{ij}^{N_{ij}}\}$ with $N_{ij} = |D_{ij}|$ be a finite set of extracted think time delays (in milliseconds), for a transition from a Markov State s_i to a Markov State s_j. For calculating a think time that follows a normal distribution, the mean μ_{ij} and the standard deviation σ_{ij} can be determined in the following way [Kirkup and Frenkel, 2006]: the mean for the elements of D_{ij} is defined as:

$$\mu_{ij} = \frac{\sum_{k=1}^{N_{ij}} d_{ij}^k}{N_{ij}}$$

As an intermediate result, the variance ν_{ij} of the mean can be obtained:

$$\nu_{ij} = \frac{\sum_{k=1}^{N_{ij}} (d_{ij}^k - \mu_{ij})^2}{N_{ij}}$$

The standard derivation denotes the square root of the variance:

$$\sigma_{ij} = \sqrt{\nu_{ij}}$$

The Behavior Model Extractor has been extended for calculating mean and standard deviation that way. Think times are written –additionally to transition probabilities– into the CSV files that include Behavior Models. The new file format will be introduced in context of additional implementations for the Markov4JMeter add-on, which will be discussed in the next section.

5.2 Think Time Emulation Support in Markov4JMeter

The (virtual) user's workload which is formalized by a Markov4JMeter workload model, is generated through the Markov4JMeter add-on for JMeter. This add-on supports the generation of probabilistic request sequences, as indicated by Behavior Models. However, think times have not been considered up to Markov4JMeter version 1.0.20080617 yet. The remainder of this section describes the add-on support for normally distributed think times, which has been implemented for this thesis.

Section 5.2.1 gives an overview of the extended Markov4JMeter add-on's functionality. Section 5.2.2 explains how think times can be defined for a workload model. Section 5.2.3 outlines the changes which have been made in the add-on's architecture. Section 5.2.4 illustrates the extensibility of the add-on in terms of various think time types. Section 5.2.5 discusses limitations and open issues for future work.

5.2.1 Overview

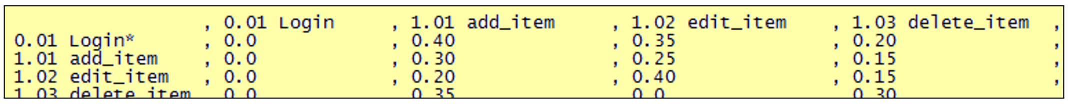
Think times indicate delays which are required by the user for reaching a decision, as discussed in Section 2.3.2. Such think times follow an underlying distribution, which can be of various type, e.g., binomial, exponential, or Gaussian/normal distribution. Since think times occur between two Markov States, they are assigned to transitions; the Markov4JMeter add-on uses matrices (in CSV format) for storing the transition values of Behavior Models. In earlier versions, each entry of a matrix P denoted the probability p_{ij} of a transition from Markov State i to Markov State j. In the extended version, each entry of a matrix P represents a pair (p_{ij}, tt_{ij}), with probability p_{ij} as before and tt_{ij} defining the think time required by a user in state i to proceed to state j. Therefore, tt_{ij} needs to be defined in a specific format, which will be discussed in the following section. However, the add-on is backward-compatible, any matrices without think time definitions will be recognized as such and processed accordingly.

5.2.2 Usage

As described before, each Behavior Model is – at least – defined as a transition probability matrix, stored in a CSV file. For the additional support of user think times, each matrix entry needs to be enriched with appropriate information. Therefore, a think time definition is formatted as follows:

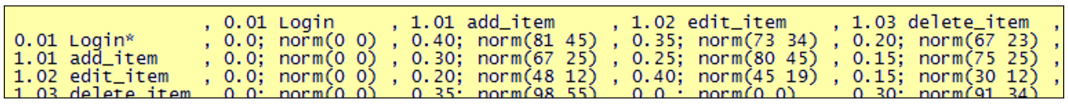
$$functionDescriptor\texttt{(}parameter_1 \;\; parameter_2 \;\; \ldots\texttt{)}$$

The $functionDescriptor$ indicates the think time distribution type to be used, whereas that type must be *unique*, that is, various think time types in one matrix are not allowed; the Markov4JMeter add-on detects such errors and gives a regarding error message in the Markov Session Controller. The number of parameters depends on that type. Figure 5.2 demonstrates the two CSV file formats for Behavior Models accepted by the



```
                    , 0.01 Login     , 1.01 add_item     , 1.02 edit_item     , 1.03 delete_item  ,
0.01 Login*         , 0.0            , 0.40              , 0.35               , 0.20              ,
1.01 add_item       , 0.0            , 0.30              , 0.25               , 0.15              ,
1.02 edit_item      , 0.0            , 0.20              , 0.40               , 0.15              ,
1.03 delete item    , 0.0            , 0.35              , 0.0                , 0.30
```

(a) Classic CSV file format for Behavior Models



```
                    , 0.01 Login     , 1.01 add_item     , 1.02 edit_item     , 1.03 delete_item  ,
0.01 Login*         , 0.0; norm(0 0) , 0.40; norm(81 45) , 0.35; norm(73 34) , 0.20; norm(67 23) ,
1.01 add_item       , 0.0; norm(0 0) , 0.30; norm(67 25) , 0.25; norm(80 45) , 0.15; norm(75 25) ,
1.02 edit_item      , 0.0; norm(0 0) , 0.20; norm(48 12) , 0.40; norm(45 19) , 0.15; norm(30 12) ,
1.03 delete item    , 0.0; norm(0 0) , 0.35; norm(98 55) , 0.0 ; norm(0 0)   , 0.30; norm(91 34)
```

(b) CSV file format for Behavior Models with (normally distributed) user think times; depicted mean and standard deviation values are of type integer, but float definitions are allowed, too.

Figure 5.2. CSV file format for Behavior Models; both figures illustrate excerpts of probability matrices for the Markov chain shown in Figure 2.1.

Markov4JMeter add-on. Figure 5.2a depicts a matrix format as used up to Markov4JMeter version 1.0.20080617, including probability values only; in Figure 5.2b, the enrichments for normally distributed think times, indicated by the descriptor *norm* are given by example. The two function parameters of each entry represent *mean* and *standard deviation* values, which will be explained in the next section. It is important to note that think times of transitions, whose destination state is the exit state, are irrelevant, since those transitions are not associated with any think times [Menascé et al., 1999]; possible definitions will be ignored by the Markov4JMeter add-on.

Normal Think Time Distribution Currently, the only supported think time distribution is of type *normal distribution*, which builds on a mean value μ and a standard deviation σ; it indicates a symmetrical curve about μ as a central value with the following properties [Wenclawiak et al., 2010]:

▻ 68.27% of the values are in the range $\mu \pm 1 \cdot \sigma$

▻ 95.45% of the values are in the range $\mu \pm 2 \cdot \sigma$

▻ 99.73% of the values are in the range $\mu \pm 3 \cdot \sigma$

Each think time delay is calculated by Markov4JMeter in accordance to the formula

$$delay = \mu + factor * \sigma * r,$$

whereas μ and σ denote the mean respectively standard deviation values, $factor$ is a constant value in $[0, 1]$, and r is an individual (Gaussian) random value with mean 0.0 and standard deviation 1.0; in particular, r might lie beyond $[0, 1]$, implying that $delay$ becomes negative. In such cases, $delay$ is set to 0, as negative think times are not allowed; to decrease the rate of negative occurrences, the constant $factor$ might be set appropriate, following the properties of a normal distribution as stated above. For example, changing $factor$ from its standard value 1.0 to 0.5, implies that think times will be corrected in much fewer cases.

5.2.3 Architecture

This section illustrates the changes of the Markov4JMeter add-on, that have been made for implementing think time support. Figure 5.3 gives an overview of the classes which have been integrated to the existing code. The individual class purposes are as follows:

▻ *BehaviorMixEntry* is a Markov4JMeter class, which processes matrix entries of Behavior Models during an experiment; therefore, it needs to detect think times and parse them, if available. For this purpose, a dedicated parser instance is contained in that class.

▻ *ThinkTimeParser* is intended for parsing CSV file tokens that represent user think times. The *parse()* method returns a valid object that represents a given think time, or null if parsing fails.

▻ *ThinkTime* is the (abstract) base class of all think time types, which follow a corresponding distribution.

▻ *NormallyDistributedThinkTime* represents, as indicated by its name, a normally distributed think time and calculates corresponding delay values. Computation of those values is based on random values, generated by the `nextGaussian()` method of the `java.util.Random` class. The code for specifying the $factor$ value discussed in the preceding section, is also included in this class. Mean and standard deviation values denote milliseconds; however, they are stored as double values, to allow more precise definitions. The method *getDelay()* returns the computed delay in milliseconds accordingly.

5.2.4 Extensibility

The extensibility of the think time related part of the Markov4JMeter add-on can be focused on the implementation of new think time types. For that purpose, an appropriate subclass

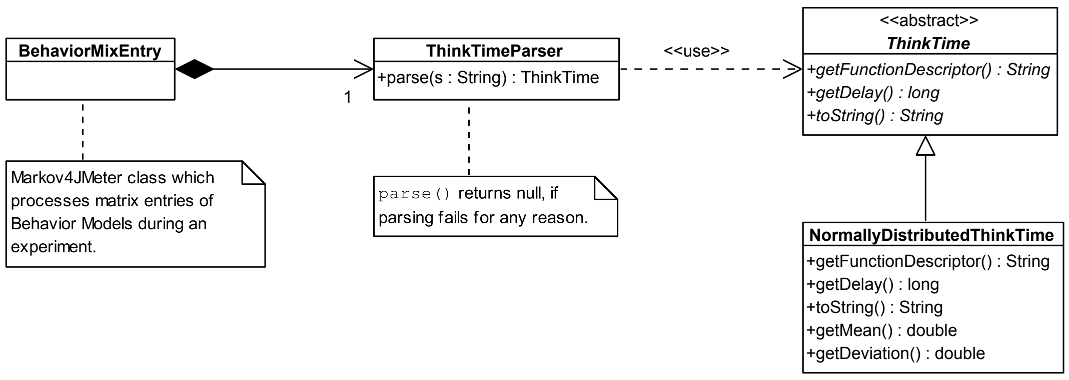

BehaviorMixEntry
1
ThinkTimeParser
+parse(s : String) : ThinkTime
<<use>>
<<abstract>>
ThinkTime
+getFunctionDescriptor() : String
+getDelay() : long
+toString() : String
Markov4JMeter class which processes matrix entries of Behavior Models during an experiment.
parse() returns null, if parsing fails for any reason.
NormallyDistributedThinkTime
+getFunctionDescriptor() : String
+getDelay() : long
+toString() : String
+getMean() : double
+getDeviation() : double

Figure 5.3. Class (sub-)model for user think time support in Markov4JMeter.

of *ThinkTime* must be added to the system. Furthermore, the new type must be registered in the *ThinkTimeParser* class, to enable parsing of related values.

5.2.5 Limitations and Future Work

This section discusses the open issues which have been identified regarding to the Markov4JMeter add-on.

- ▷ The lack of alternative think times distributions denotes a general open issue; however, for tests which include normal think time distributions only, the existing think time type should be sufficient. For any new types, the add-on can be easily extended, as described in Section 5.2.4.
- ▷ A measurement-based evaluation for analyzing the impact of updated think time emulation on test results would be desirable. Think times for classic Behavior Models have been already modeled in context of the research project DynaMod, and that modeling approach could be used for comparison purposes.

Chapter 6

Evaluation

This chapter presents an evaluation of the approach for integrating tests into a generative platform, as introduced in the preceding chapters. Section 6.1 starts with the qualitative aspects, illustrating the current status of the JMeter Test Plan generation process. Since the process still reveals several open issues, focus will be put on them. Section 6.2 discusses quantitative aspects, and shows that the generated Test Plan cover a much higher amount of scenarios as manual constructed ones.

6.1 Qualitative Evaluation

As a proof of concept, the generation of a JMeter Test Plan for the b+m gear application *CarShare* will be considered in this section. That system has been introduced in Section 2.5.3, and its generation process provides a set of Screenflows, which can be transformed to the core structure of a corresponding JMeter Test Plan. Section 6.1.1 presents the structure of a Test Plan, which results from the transformation framework; thereby, the working parts of the approach will be illustrated. Section 6.1.2 discusses open issues and gives recommendations for possible solutions.

6.1.1 Test Plan Generation for CarShare

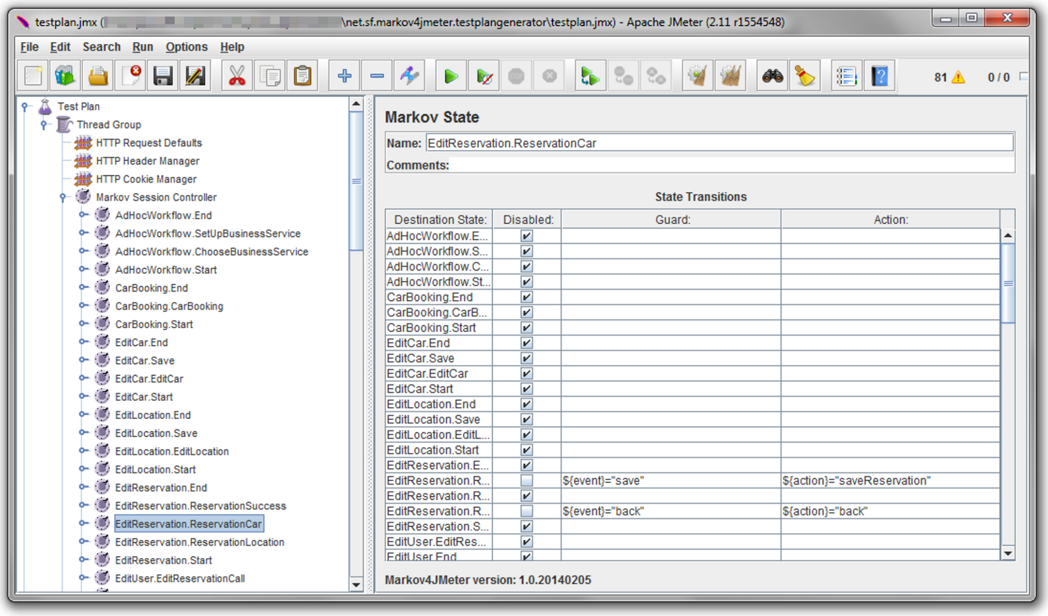
For the transformation of CarShare Screenflows to a corresponding Test Plan structure, the following three steps have been performed:

1. Interception of Screenflow information from the CarShare generation process, as described in Section 4.1.2.

2. Transformation of the obtained Screenflows to a corresponding M4J-DSL model, by using the M4J-DSL Model Generator introduced in Section 4.1.

3. Transformation of the M4J-DSL model to a JMeter Test Plan, by using the Test Plan Generator introduced in Section 3.2.

Figure 6.1 shows the resulting Test Plan of this process. The structure includes HTTP elements by default, which can be removed or substituted with alternative elements in JMeter manually. At this point, a DSL for a Test Plan structure, as discussed in Section 3.2.5,



testplan.jmx ([redacted]\net.sf.markov4jmeter.testplangenerator\testplan.jmx) - Apache JMeter (2.11 r1554548)

- Test Plan
  - Thread Group
    - HTTP Request Defaults
    - HTTP Header Manager
    - HTTP Cookie Manager
    - Markov Session Controller
      - AdHocWorkflow.End
      - AdHocWorkflow.SetUpBusinessService
      - AdHocWorkflow.ChooseBusinessService
      - AdHocWorkflow.Start
      - CarBooking.End
      - CarBooking.CarBooking
      - CarBooking.Start
      - EditCar.End
      - EditCar.Save
      - EditCar.EditCar
      - EditCar.Start
      - EditLocation.End
      - EditLocation.Save
      - EditLocation.EditLocation
      - EditLocation.Start
      - EditReservation.End
      - EditReservation.ReservationSuccess
      - EditReservation.ReservationCar
      - EditReservation.ReservationLocation
      - EditReservation.Start
      - EditUser.EditReservationCall

Markov State

Name: EditReservation.ReservationCar

Comments:

State Transitions

Destination State:	Disabled:	Guard:	Action:
AdHocWorkflow.E...	✓		
AdHocWorkflow.S...	✓		
AdHocWorkflow.C...	✓		
AdHocWorkflow.St...	✓		
CarBooking.End	✓		
CarBooking.CarB...	✓		
CarBooking.Start	✓		
EditCar.End	✓		
EditCar.Save	✓		
EditCar.EditCar	✓		
EditCar.Start	✓		
EditLocation.End	✓		
EditLocation.Save	✓		
EditLocation.EditL...	✓		
EditLocation.Start	✓		
EditReservation.E...	✓		
EditReservation.R...	☐	${event}="save"	${action}="saveReservation"
EditReservation.R...	✓		
EditReservation.R...	☐	${event}="back"	${action}="back"
EditReservation.S...	✓		
EditUser.EditRes...	✓		
EditUser.End	✓		

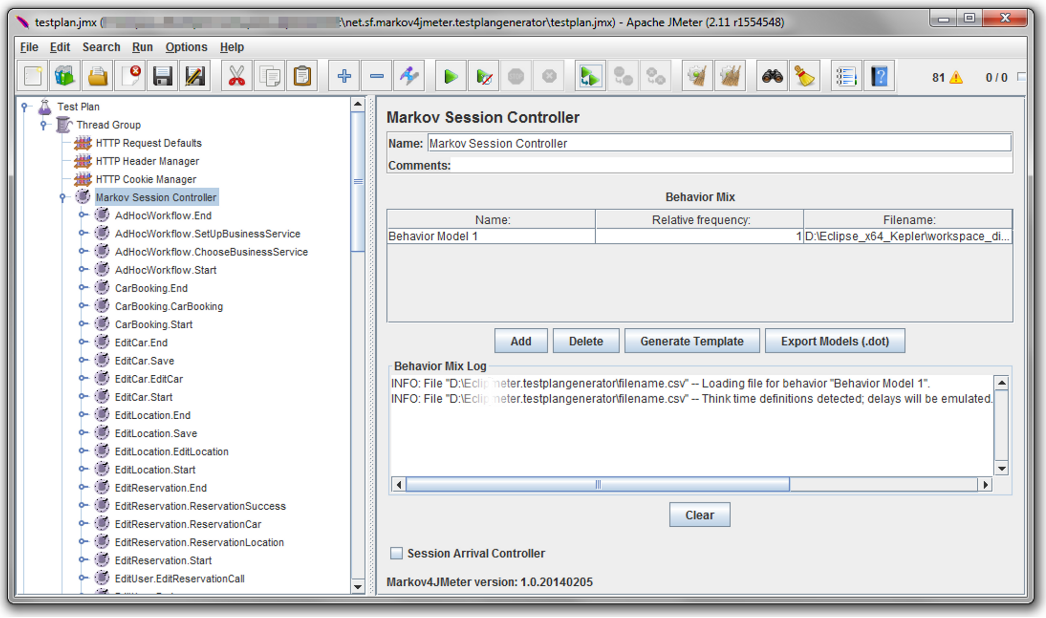
Markov4JMeter version: 1.0.20140205

Figure 6.1. Generated JMeter Test Plan for the b+m gear application CarShare (excerpt).

appears to be helpful for predefining an appropriate structure. The Test Plan also includes the Markov Session Controller with Markov States as child nodes; each of those states represents a CarShare View. The underlying Protocol Layer EFSMs, modeled as children of Markov States, are hidden, since they denote generic requests only. The configuration panel on the right side of the screenshot depicts the preconfigured input options of a Markov State. Those include the definition of (valid) destination states, which indicate outgoing transitions with their guards and actions. Figure 6.2 shows the configuration panel for the Markov Session Controller of a generated Test Plan. The *Behavior Mix Log* textfield, which has been added within the scope of this thesis, indicates successful loading of (generated) Behavior Models, and sucessful detection of think time definitions. However, trying to run a test for such a generated Test Plan has no further effect yet, since the underlying Protocol Layer information is still missing. Therewith, the Test Plan is not applicable for an experiment currently. The following section illustrates the open issues, which need to be solved for obtaining the required protocol information.

6.1.2 Open Issues and Recommendations

For generating Test Plans, which can be run for a JMeter experiment, additional adjustments need to be made in context of the actual Test Plan generation and JMeter, as well as for the mapping between Markov4JMeter and b+m gear models. The required implementations



testplan.jmx ([redacted]\net.sf.markov4jmeter.testplangenerator\testplan.jmx) - Apache JMeter (2.11 r1554548)

Markov Session Controller

Name: Markov Session Controller

Comments:

Behavior Mix

Name:	Relative frequency:	Filename:
Behavior Model 1	1	D:\Eclipse_x64_Kepler\workspace_di...

Behavior Mix Log

```
INFO: File "D:\Eclip   neter.testplangenerator\filename.csv" -- Loading file for behavior "Behavior Model 1".
INFO: File "D:\Eclip   neter.testplangenerator\filename.csv" -- Think time definitions detected; delays will be emulated.
```

☐ Session Arrival Controller

Markov4JMeter version: 1.0.20140205

Figure 6.2. Markov Session Controller of a generated Test Plan; file paths in the *Behavior Mix Log* textfield have been shortened for the sake of clarity.

will be discussed in this section.

Test Plan Generation and JMeter

▻ *GOTO Controllers*. The non-availability of *GOTO Controllers* in JMeter constitutes a strong restriction for modeling Protocol Layer EFSMs, which formalize the protocol-specific control flows of Markov States. Equivalent IF/WHILE structures, as discussed in Section 3.2.5, might denote a workaround. Since those structures are hard to maintain, a preferable solution would be the implementation of a dedicated GOTO Controller for JMeter. Such a controller should make it possible to model arbitrary branches between Test Plan Elements, e.g., by offering a *destination element* field for an element name or ID. This seems to be solvable, as the Markov4JMeter add-on also influences the control flow of JMeter: when a Test Plan is processed, the Markov Session Controller branches to a (randomly selected) Markov State element.

▻ *Guards and actions in Markov4JMeter*. Guards and actions in Markov4JMeter are evaluated as JavaScript expressions. Hence, their scope is limited to the underlying JavaScript engine. For using Markov4JMeter as a locally running testing engine, based on Java requests, more flexibility, e.g., call of Java functions for an action, would be of great help. Furthermore, guards and actions in Markov4JMeter denote conditional expressions; this differs from certain models –such as in b+m gear– which, e.g., use actions as indicators for method calls.

b+m gear recommendations

▻ In context of UI tests in b+m gear, ID handling in Vaadin 7 still constitutes an open issue. As a workaround, a dedicated servlet has been released by Vaadin Ltd. [2014]. However, this does not work consistently yet. The fact that corresponding tickets exist in the developer's open tracking system, indicates that the problem has been faced. Currently, the only alternative solution consists of code analysis, to possibly modify the synchronization mechanism of a Vaadin system accordingly.

▻ Java applications to be tested must be located as JAR file in the installation path of JMeter. This is difficult to be solved for Web applications like CarShare, as such systems are initialized through the Web environment which, e.g., initializes the connection to a database. Hence, service functions cannot be simple called as single Java methods. For a proper use of JMeter for emulating user behavior, the flow controller of the front-end must be "emulated" by an own framework, if the front-end layer shall be circumvented.

▻ To facilitate JMeter testing, b+m gear might provide additional code stubs to be utilized during test- or runtime. The facilitation through the Service Layer with regard to tests should constitute a general design aspect for the platform.

▻ The protocol information required for Test Plans must be provided in the generation process of gear. Therefore, the input data which is accepted by a View, must be specified. This denotes a challenge in context of input fields with inter-dependencies (e.g., an item of a combobox that is only selectable, if another widget has a certain input value). Generally, model annotations for Views might help to specify at least the "type" of input value. Inter-dependencies might be specified in a similar way.

▻ The monitoring must be adjusted for the Views. The challenge thereby is the identification of View in the monitored user session traces. This might be solved through the use of transition models, as constructed in context of DynaMod, as described in [Schulz, 2013]. It must be ensured that View sessions can be reconstructed that way, for analyzing session traces accordingly. Thereby, a particular challenge is the non-availability of server events, as the front-end partition is not involved.

6.2 Quantitative Evaluation

Although application-related Protocol Layer EFSMs are not included in the generated JMeter Test Plans yet, a quantitative evaluation in terms of time saving aspects can be made already. In context of the research project *DynaMod* [van Hoorn et al., 2011, 2013], hand-crafted JMeter Test Plans have been used for testing a b+m gear application, shortly

Table 6.1. Quantitative comparison between generated (CarShare) and manually constructed (b+m gear-AIDA) JMeter Test Plans; the estimated time needed for constructing a Test Plan refers to the Session Layer creation only.

	CarShare	**b+m gear-AIDA**
# Application States	82	37
# transitions	192	–
Test coverage (estimated)	100%	25%
Time needed (estimated)	1–3 day (environment setup)	2–3 weeks (incl. intermediate tests)

referred to as *b+m gear-AIDA*. This application is very similar to the CarShare application, hence it constitutes a good basis for a comparison between the effort for constructing Test Plans manually and the effort for generating Test Plans.

Case Study For the CarShare application, 21 Screenflow are obtainable from its generation process. The Test Plan generator transforms these Screenflows to a JMeter Test Plan of 82 Application States and 192 transitions. The amount of these elements is far beyond the amount of hand-crafted elements included in the b+m gear-AIDA Test Plan. For b+m gear-AIDA, transitions of the Session Layer, as depicted in Figure 6.2, have not been modeled individually for time saving purposes, assuming that all transitions are valid.

Table 6.1 gives a quantitative comparison between the generated and manually constructed JMeter Test Plans. The estimated needed time does not include the time for familiarization. This would even increase the time range for manual construction, as JMeter elements generally provide many configuration options.

Chapter 7

Related Work

This section gives a short overview of related work, which mainly aims to the modeling of workload intensity, as well as to the open issues concerning AJAX-systems.

- ▷ The work of v. Kistowski et al. [2014] aims to the flexible definition of load profiles, whose support is only limited in conventional benchmarking frameworks [v. Kistowski et al., 2014]. Therefore, the authors use two meta-models on different abstraction levels for describing the load intensity: the *Descartes Load Intensity Meta-Model* and the *High-Level Descartes Load Intensity Meta-Model*. The first model allows to define load intensity over time via mathematical functions; with the second model, load intensity can be defined by the use of parameters, which characterize seasonal patterns, trends, bursts and noise parts [v. Kistowski et al., 2014]. To facilitate the handling of the two models, a toolkit named *LIMBO* is provided. With that tool, formulas which describe the workload intensity might be created for being used in combination with the JMeter add-on. This denotes an open issue for future work, as described in Section 6.1.2.
- ▷ Blum [2013] introduces an alternative approach for the generation of performance tests from a Web application. This work provides a tool named *Lightning*, which uses a Web crawler in combination with an HTTP proxy server, for the generation of interaction models for the targeted systems. Based on stochastic methods, Lightning creates complete JMeter Test Plans which emulate the user behavior. In particular, it also considers AJAX systems. The underlying idea differs from the approach followed in this thesis, since Test Plans are not generated from underlying models of MDSD applications. However, it possibly provides additional know-how for the generation of JMeter Test Plans particularly for AJAX-based systems.

Chapter 8

Conclusions and Outlook

This chapter draws the conclusion and gives an outlook to future work. Section 8.1 summarizes the results of this thesis. Section 8.2 discusses the identified difficulties and open issues as well. Section 8.3 presents several issues which might indicate a direction future work.

8.1 Summary

This work introduced an approach for integrating the generation of JMeter Test Plans, which target MDSD applications, in a generative platform named *b+m gear*. The approach builds on Screenflow definitions, that are given through the underlying model of a targeted application. The implementation has been divided into three parts:

▷ At first, a domain-specific language, namely *M4J-DSL*, has been defined for building model instances, that represent workload models for Markov4JMeter. Such models formalize the (virtual) user's workload to be generated through the Markov4JMeter add-on for JMeter. To transform them into corresponding Test Plans, a standalone framework, referred to as *Test Plan Generator*, has been realized. The requirement of input model validation, to detect possible inconsistencies on start of a transformation process, is fully supported by this framework.

▷ Second, the transformation of Screenflow information into M4J-DSL models has been realized. Screenflows can be obtained through the generation process of a b+m gear application. For their transformation into a corresponding M4J-DSL model, the *M4J-DSL model generator* has been implemented. That generator additionally includes probabilities and think times, possibly resulting from monitoring data, into the Behavior Models of an M4J-DSL model.

▷ As a third part, the extraction of Behavior Models from monitoring data has been extended by the support of user think time extraction. Furthermore, think time support has been added to the Markov4JMeter add-on, by extending the CSV file format appropriately, and implementing the emulation of delays accordingly.

In the second implementation part, it turned out that the generation of JMeter Test Plans in terms of performance tests currently fails for two reasons: first, the technical realization for

AJAX-based systems depends on technical conditions, which are not fulfilled through the Vaadin 7 UI; this has been already illustrated in Section 2.6.3. Second, the transformation of Screenflows into corresponding Test Plans is complicated by fundamental mismatches between the b+m gear and Markov4JMeter models, as discussed in Section 6.1.2.

For the first issue, the original focus put on performance testing has turned an alternative approach, using JMeter as a flow control engine, to circumvent the UI part of the b+m gear model's Frontend partition. However, this holds new challenges, since underlying techniques of the UI need to be emulated, which denotes a non-trivial task. An according implementation would have exceeded the scope of this thesis; hence, recommendations have been given in the evaluation, as well as for the second issue. The quantitative part of the evaluation illustrated the benefits obtained from generated (partial) Test Plans, with regard to time-saving and test coverage issues.

8.2 Discussion

The integration of performance tests into a generate platform, which denoted the main goal of this thesis, has not been reached completely. The reason is mainly given through technical issues, which constitute critical aspects to be considered more closely.

The identified challenges concerning AJAX technologies raise the question, whether the UI part of an application generally should be included in tests which aim to the performance analysis of certain service functions. UI frameworks like Vaadin tend to run background tasks for optimization purposes on server-side, which make the identification of possible malfunctions caused by service methods difficult. It might remain unclear whether unexpected behavior, e.g., bottlenecks, result from the UI or from an underlying service method. To circumvent such ambiguities, an engine which aims to the invocation of service functions, by emulating behavioral patterns of real users, constitutes a solution. Such tool is not limited to performance tests, it can be used for regression tests as well. However, it does not solve the open issues concerning AJAX-based systems, which constitute general challenges in terms of performance testing.

It turned out that the underlying technology for AJAX-based systems hangs by a thread, meaning that the possibility to test such applications mainly depends on the control options for the synchronization between clients and server. In times of Web 2.0, this is a critical issue, since the complexity of Web applications increases rapidly, while performance testing tools like JMeter do not keep pace with this development. A significant barrier is the missing JavaScript support in JMeter, as illustrated in Section 2.6.2. Even if JavaScript is supported, the identification of proper function calls remains a difficult task, including code analysis which is possibly made more difficult through obfuscation techniques. However, the support of JavaScript would offer approaches to new goals.

It has proved in the evaluation part of this thesis, that the generation of Test Plans is –at least– partially possible, but additional implementations are required for completing the approach. The benefits of a completion are obviously: as shown in the quantitative part of the evaluation, time-savings are enormous, which denotes a crucial economic factor for companies. Furthermore, it has been proved through the currently obtainable transformation results, that models indeed can provide information for generating Test Plans without the drawbacks of fragility and limited coverage of test scenarios, as discussed in Section 1.2. Hence, the completion of this approach should be followed in future research.

8.3 Future Work

The future work includes implementation issues and evaluation issues as well. Open implementation issues for completing the Test Plan generation framework have been already discussed in the corresponding implementation chapters of this thesis. They mostly appear to be solvable, even though the additional effort might be high. Even if the framework would be completed as discussed before, the following enhancements –possibly under use of external frameworks– are desirable:

- *Inclusion of realistic workload intensity*. Generated Test Plans should be equipped with proper workload intensity formulas for the Markov Session Controller of the Markov4-JMeter add-on; currently, such formulas are defined manually for the generator, without connection to the actual workload of the SUT. The work of v. Kistowski et al. [2014] provides helpful tools to solve this issue; this will be further discussed in Chapter 7.

- *Clustering methods for Behavior Models*. The number of Behavior Models, respectively associated user types, is still not determinable by the framework; currently, it is assumed that exactly one user behavior type exists. This can be solved by the use of clustering methods, as already discussed by Schulz [2013].

In case the Test Plan generation process can be completed that way, the validity of (generated) load test scripts constitutes the only major issue to be solved in context of the performance testing challenges discussed for DynaMod in Section 1.2. For a measurement-based evaluation, a system in productive operation should be considered, to prove that results of performance tests using generated Test Plans are reasonable. A particular goal should be to demonstrate that those Test Plans provide a much higher coverage of scenarios, higher robustness, and –with appropriate default value handlers in particular– more flexibility as hand-crafted scripts.

Appendix A

M4J-DSL Validation Constraints

The following 34 constraints have been identified for validating M4J-DSL models. The meta-model introduced in Section 3.1 has been enrichted with these constraints, either by implementing them via oclInEcore editor or by setting properties of certain class attributes appropriately. In particular, `oclIsUndefined()` contraints can be simply implemented in Ecore by setting the lower and upper bounds of the regarding class attributes both to 1. Constraints associated with the same context might be AND-linked; however, they are listed individually since they have been implemented as well for detecting violations more precisely.

1. The formula for workload intensity must be defined:

```
context WorkloadIntensity
inv: not formula->oclIsUndefined()
```

2. The (constant) number of sessions must be nonnegative:

```
context ConstantWorkloadIntensity
inv: numberOfSessions >= 0
```

3. The states list of a Session Layer EFSM must include the initial state:

```
context SessionLayerEFSM
inv: not initialState.oclIsUndefined()
    implies applicationStates->includes(initialState)
```

4. The states list of a Protocol Layer EFSM must include the initial state:

```
context ProtocolLayerEFSM
inv: not initialState.oclIsUndefined()
    implies protocolStates->includes(initialState)
```

5. The states list of a Behavior Model must include the initial state:

```
context BehaviorModel
inv: not initialState.oclIsUndefined()
    implies markovStates->includes(initialState)
```

6. Session Layer EFSM states must be associated with unique services:

```
context SessionLayerEFSM
inv: applicationStates->forAll(s1,s2 |
    (s1 <> s2 and not(
        s1.service.oclIsUndefined() or s2.service.oclIsUndefined()
    )) implies s1.service <> s2.service)
```

7. Protocol Layer EFSM states must be associated with unique requests:

```
context ProtocolLayerEFSM
inv: protocolStates->forAll(s1,s2 |
    (s1 <> s2 and not(
        s1.request.oclIsUndefined() or s2.request.oclIsUndefined()
    )) implies s1.request <> s2.request)
```

8. Protocol Layer EFSMs must contain their own states only:

```
context ProtocolLayerEFSM
inv: protocolStates->forAll(s |
    s.outgoingTransitions->forAll(t |
        not t.targetState.oclIsUndefined()
            implies (protocolStates->includes(t.targetState) or
                t.targetState = exitState)))
```

9. Behavior Models must contain their own states only:

```
context BehaviorModel
inv: markovStates->forAll(s |
    s.outgoingTransitions->forAll(t |
        not t.targetState.oclIsUndefined()
            implies (markovStates->includes(t.targetState) or
                t.targetState = exitState)
```

10. Outgoing transitions of an Application State must be unique:

```
context ApplicationState
inv: outgoingTransitions->forAll(t1,t2 |
    (t1 <> t2 and not(
        t1.targetState.oclIsUndefined() or t2.targetState.oclIsUndefined()
    )) implies (t1.targetState <> t2.targetState or t1.guard <> t2.guard))
```

11. Outgoing transitions of a Protocol State must be unique:

```
context ProtocolState
inv: outgoingTransitions->forAll(t1,t2 |
    (t1 <> t2 and not(
        t1.targetState.oclIsUndefined() or t2.targetState.oclIsUndefined()
    )) implies (t1.targetState <> t2.targetState or t1.guard <> t2.guard))
```

12. The name of a service must be defined:

```
context Service
inv: not name->oclIsUndefined()
```

13. The name of a service must be unique:

```
context Service
inv: Service.allInstances()->forAll(s1,s2 |
    (s1 <> s2 and not(s1.name.oclIsUndefined() or s2.name.oclIsUndefined()))
        implies s1.name <> s2.name)
```

14. The test-pattern of an assertion must be defined:

```
context Assertion
inv: not patternToTest->oclIsUndefined()
```

15. All requests must be of same type:

```
context Request
inv: Request.allInstances()->forAll(r1,r2 | r1.oclType() = r2.oclType())
```

16. The properties of a request must have unique keys:

```
context Request
inv: properties->forAll(p1,p2 |
    (p1 <> p2 and not(p1.key.oclIsUndefined() or p2.key.oclIsUndefined()))
        implies p1.key <> p2.key)
```

17. The parameters of a request must have unique names:

```
context Request
inv: parameters-> forAll(p1,p2 |
    (p1 <> p2 and not(p1.name.oclIsUndefined() or p2.name.oclIsUndefined()))
        implies p1.name <> p2.name)
```

18. The key of a property must be defined:

```
context Property
inv: not key->oclIsUndefined()
```

19. The name of a parameter must be defined:

```
context Parameter
inv: not name->oclIsUndefined()
```

20. The sum of relative frequencies in a Behavior Mix must be 1.0:

```
context BehaviorMix
inv: relativeFrequencies.value->sum() = 1.0
```

21. Relative frequencies must be assigned to unique Behavior Models in a Behavior Mix:

```
context BehaviorMix
inv: relativeFrequencies->forAll(f1,f2 |
    (f1 <> f2 and not(
        f1.behaviorModel.oclIsUndefined() or f2.behaviorModel.oclIsUndefined()
    )) implies f1.behaviorModel <> f2.behaviorModel)
```

22. Relative frequencies must range between 0.0 and 1.0:

```
context RelativeFrequency
inv: value >= 0.0 and value <= 1.0
```

23. The name of a Behavior Model must be defined:

```
context BehaviorModel
inv: not name->oclIsUndefined()
```

24. The name of a Behavior Model must be unique:

```
context BehaviorModel
inv: BehaviorModel.allInstances()->forAll(b1,b2 |
    (b1 <> b2 and not(b1.name.oclIsUndefined() or b2.name.oclIsUndefined()))
        implies b1.name <> b2.name)
```

25. The filename assigned to a Behavior Model must be defined:

```
context BehaviorModel
inv: not filename->oclIsUndefined()
```

26. The filename assigned to a Behavior Model must be unique:

```
context BehaviorModel
inv: BehaviorModel.allInstances()->forAll(b1,b2 |
    (b1 <> b2 and not(
        b1.filename.oclIsUndefined() or b2.filename.oclIsUndefined()
    )) implies b1.filename <> b2.filename)
```

27. Behavior Models must include Markov States for all services:

```
context BehaviorModel
inv: Service.allInstances()->
    forAll(s | markovStates->exists(m | m.service = s))
```

28. The sum of probabilities of all outgoing transitions in a Markov State must be 1.0:

```
context MarkovState
inv: outgoingTransitions.probability->exists(p | p > 0) implies
    outgoingTransitions.probability->sum() = 1.0
```

29. The Markov States must be associated with unique services:

```
context MarkovState
inv: MarkovState.allInstances()->forAll(s1,s2 |
    (s1 <> s2 and not(
        s1.service.oclIsUndefined() or s2.service.oclIsUndefined()
    )) implies s1.service <> s2.service)
```

30. Outgoing transitions of a Markov State state must have unique target states:

```
context MarkovState
inv: outgoingTransitions->forAll(t1,t2 |
    (t1 <> t2 and not(
        t1.targetState.oclIsUndefined() or t2.targetState.oclIsUndefined()
    )) implies t1.targetState <> t2.targetState)
```

31. Outgoing transitions of a Markov State state must correspond to the Session Layer:

```
context MarkovState
inv: not service.oclIsUndefined()
    implies ApplicationState.allInstances()->exists(as |
        service = as.service and outgoingTransitions->forAll(t |
            not t.targetState.oclIsUndefined()
                implies as.outgoingTransitions->exists(at |
                    (at.targetState.oclIsTypeOf(ApplicationExitState) and
                      t.targetState.oclIsTypeOf(BehaviorModelExitState)) or
                    ((t.targetState.oclIsTypeOf(MarkovState) and
                      at.targetState.oclIsTypeOf(ApplicationState) and
                     at.targetState.oclAsType(ApplicationState).service =
                      t.targetState.oclAsType(MarkovState).service)))))
```

32. The probability of a Markov Transition must range between 0.0 and 1.0:

```
context Transition
inv: probability >= 0.0 and probability <= 1.0
```

33. Markov Transitions must have think times of same type:

```
context Transition
inv: Transition.allInstances()->forAll(t1,t2 |
    not (t1.thinkTime.oclIsUndefined() or t2.thinkTime.oclIsUndefined())
        implies t1.thinkTime.oclType() = t2.thinkTime.oclType())
```

34. Normally distributed think times must be valid:

```
context NormallyDistributedThinkTime
inv: mean >= 0.0 and deviation >= 0.0 and deviation <= mean
```

Table A.1. Constraint violation messages for the M4J-DSL. The indexes in the first column comply to the indexes assigned to the constraints listed above. Constraints with no assigned messages have been implemented without using OCL expressions; the notification for any of their violations is EMF-dependent. Equal messages can be distinguished by their related context.

Constraint	Assigned Message	Context
1	–	WorkloadIntensity
2	mustBeNonnegativeSessionNumber	ConstantWorkloadIntensity
3	mustBeInitialStateWhichIsIncludedInApplicationStatesList	SessionLayerEFSM
4	mustBeInitialStateWhichIsIncludedInProtocolStatesList	ProtocolLayerEFSM
5	mustBeInitialStateWhichIsIncludedInMarkovStatesList	BehaviorModel
6	mustBeApplicationStatesWithUniqueServices	SessionLayerEFSM
7	mustBeProtocolStatesWithUniqueRequests	ProtocolLayerEFSM
8	mustBeProtocolLayerEFSMWithoutForeignStates	ProtocolLayerEFSM
9	mustBeBehaviorModelWithoutForeignTargetStates	BehaviorModel
10	mustBeUniqueOutgoingTransitions	ApplicationState
11	mustBeUniqueOutgoingTransitions	ProtocolState
12	–	Service
13	mustBeUniqueNames	Service
14	–	Assertion
15	mustBeRequestsOfSameType	Request
16	mustBeUniquePropertyKeys	Request
17	mustBeUniqueParameterNames	Request
18	–	Property
19	–	Parameter
20	mustBeValidFrequencySum	BehaviorMix
21	mustBeUniqueBehaviorModels	BehaviorMix
22	mustBeValidFrequency	RelativeFrequency
23	–	BehaviorModel
24	mustBeUniqueNames	BehaviorModel
25	–	BehaviorModel
26	mustBeUniqueFilenames	BehaviorModel
27	mustBeBehaviorModelWithMarkovStatesForAllServices	BehaviorModel
28	mustBeValidProbability	MarkovState
29	mustBeMarkovStatesWithUniqueServices	MarkovState
30	mustBeOutgoingTransitionsWithUniqueTargetStates	MarkovState
31	mustBeOutgoingTransitionsCorrespondingToSessionLayer	MarkovState
32	mustBeValidProbabilitySum	Transition
33	mustBeThinkTimesOfSameType	Transition
34	mustBeValidThinkTimeValues	NormallyDistributedThinkTime

Appendix B

Supported JMeter Test Plan Elements

Table B.1. JMeter Test Plan elements supported by the Test Plan Generator. The elements are classified into categories which comply to those used in the JMeter application. The element named *Test Plan* is displayed at first, since it does not belong to any specific category. Both supported *Listeners* correspond to the same Java class, differing only in the various GUI representations in JMeter.

Name of JMeter Element	**Corresponding Java Class**
Test Plan	`org.apache.jmeter.testelement.TestPlan`
Assertions	
Response Assertion	`org.apache.jmeter.assertions.ResponseAssertion`
Config Elements	
Cookie Manager	`org.apache.jmeter.protocol.http.control.CookieManager`
Counter	`org.apache.jmeter.modifiers.CounterConfig`
HTTP Header Manager	`org.apache.jmeter.protocol.http.control.HeaderManager`
HTTP Request Defaults	`org.apache.jmeter.config.ConfigTestElement`
User Defined Variables	`org.apache.jmeter.config.Arguments`
Listeners	
Response Time Graph	`org.apache.jmeter.reporters.ResultCollector`
View Results Tree	`org.apache.jmeter.reporters.ResultCollector`
Logic Controllers	
If Controller	`org.apache.jmeter.control.IfController`
Loop Controller	`org.apache.jmeter.control.LoopController`
While Controller	`org.apache.jmeter.control.WhileController`
Markov Session Controller	`net.voorn.markov4jmeter.control.MarkovController`
Markov State	`net.voorn.markov4jmeter.control.ApplicationState`
Post Processors	
Regular Expression Extractor	`org.apache.jmeter.extractor.RegexExtractor`
Samplers	
HTTP Request	`org.apache.jmeter.protocol.http.sampler.HTTPSamplerProxy`
SOAP/XML-RPC Request	`org.apache.jmeter.protocol.http.sampler.SoapSampler`
Java Request	`org.apache.jmeter.protocol.java.sampler.JavaSampler`
JUnit Request	`org.apache.jmeter.protocol.java.sampler.JUnitSampler`
BeanShell Sampler	`org.apache.jmeter.protocol.java.sampler.BeanShellSampler`

Threads (Users)	
Setup Thread Group	`org.apache.jmeter.threads.SetupThreadGroup`

Timers	
Gaussian Random Timer	`org.apache.jmeter.timers.GaussianRandomTimer`

Appendix C

External Libraries

Table C.1. External libraries utilized by the Test Plan Generator, divided by (sub-)folder locations.

Library File	**Licence**
`commons-cli-20040117.000000.jar`	AL 2.0
jmeter/	
`avalon-framework-4.1.4.jar`	AL 2.0
`bsh-2.0b5.jar`	LGPL
`commons-httpclient-3.1.jar`	AL 2.0
`commons-io-2.4.jar`	AL 2.0
`commons-lang3-3.1.jar`	AL 2.0
`commons-logging-1.1.3.jar`	AL 2.0
`jorphan.jar`	AL 2.0
`junit-4.11.jar`	EPL
`logkit-2.0.jar`	AL 2.0
`oro-2.0.8.jar`	AL 2.0
`rsyntaxtextarea-2.5.0.jar`	BSD
`xmlpull-1.1.3.1.jar`	LGPL
`xpp3_min-1.1.4c.jar`	AL 1.1
`xstream-1.4.4.jar`	BSD
jmeter/ext/	
`ApacheJMeter_components.jar`	AL 2.0
`ApacheJMeter_core.jar`	AL 2.0
`ApacheJMeter_http.jar`	AL 2.0
`ApacheJMeter_java.jar`	AL 2.0
`ApacheJMeter_junit.jar`	AL 2.0
jmeter/ext/markov4jmeter/	
`Markov4JMeter-1.0.20140405.jar`	AL 2.0
`org.mozilla.javascript-1.7.2.jar`	AL 2.0
validation/	
`org.eclipse.ocl.ecore_3.3.0.v20130520-1222.jar`	EPL
validation/pivot/misc/	
`com.google.guava_11.0.2.v201303041551.jar`	AL 2.0
`com.google.inject_3.0.0.v201203062045.jar`	AL 2.0

org.apache.log4j_1.2.15.v201012070815.jar	AL 2.0
org.eclipse.emf.codegen_2.9.0.v20140203-1126.jar	EPL
org.eclipse.emf.common_2.9.2.v20131212-0545.jar	EPL
org.eclipse.emf.ecore.xmi_2.9.1.v20131212-0545.jar	EPL
org.eclipse.emf.ecore_2.9.2.v20131212-0545.jar	EPL
org.eclipse.xtext.common.types.ui_2.5.2.v201402120812.jar	EPL
org.eclipse.xtext.common.types_2.5.2.v201402120812.jar	EPL
org.eclipse.xtext.ui.shared_2.5.2.v201402120812.jar	EPL
org.eclipse.xtext.ui_2.5.2.v201402120812.jar	EPL
org.eclipse.xtext.util_2.5.2.v201402120812.jar	EPL
org.eclipse.xtext_2.5.2.v201402120812.jar	EPL
validation/pivot/misc/osgi/	
org.eclipse.osgi.services_3.3.100.v20130513-1956.jar	EPL
org.eclipse.osgi.util_3.2.300.v20130513-1956.jar	EPL
org.eclipse.osgi_3.9.1.v20130814-1242.jar	EPL
validation/pivot/misc/resources/	
org.eclipse.core.resources.win32.x86_64_3.5.0.v20121203-0906.jar	EPL
org.eclipse.core.resources_3.8.101.v20130717-0806.jar	EPL
org.eclipse.ui.navigator.resources_3.4.500.v20130516-1049.jar	EPL
org.eclipse.uml2.uml.resources_4.1.0.v20130902-0826.jar	EPL
validation/pivot/misc/uml2/	
com.google.guava_10.0.1.v201203051515.jar	AL 2.0
javax.inject_1.0.0.v20091030.jar	AL 2.0
org.antlr.runtime_3.2.0.v201101311130.jar	BSD
org.eclipse.emf.edit_2.9.0.v20140203-1126.jar	EPL
org.eclipse.uml2.common_1.8.1.v20130902-0826.jar	EPL
org.eclipse.uml2.types_1.1.0.v20130902-0826.jar	EPL
org.eclipse.uml2.uml.profile.l2_1.1.0.v20130902-0826.jar	EPL
org.eclipse.uml2.uml.profile.l3_1.1.0.v20130902-0826.jar	EPL
org.eclipse.uml2.uml.resources_4.1.0.v20130902-0826.jar	EPL
org.eclipse.uml2.uml_4.1.1.v20130902-0826.jar	EPL
validation/pivot/parsing/	
org.eclipse.ocl.common_1.1.0.v20130531-0544.jar	EPL
org.eclipse.ocl.examples.common_3.2.100.v20130520-1503.jar	EPL
org.eclipse.ocl.examples.domain_3.3.1.v20130817-0757.jar	EPL
org.eclipse.ocl.examples.library_3.3.1.v20130817-0632.jar	EPL
org.eclipse.ocl.examples.pivot_3.3.1.v20130817-0757.jar	EPL
org.eclipse.ocl.examples.xtext.base_3.3.1.v20130817-0929.jar	EPL
org.eclipse.ocl.examples.xtext.completeocl_3.3.0.v20130523-1559.jar	EPL
org.eclipse.ocl.examples.xtext.essentialocl_3.3.1.v20130817-0639.jar	EPL
org.eclipse.ocl.examples.xtext.oclinecore_3.3.0.v20130527-1543.jar	EPL
org.eclipse.ocl.examples.xtext.oclstdlib_3.3.0.v20130523-1559.jar	EPL
validation/pivot/ui/	
org.eclipse.ocl.common.ui_1.1.0.v20130530-0730.jar	EPL

org.eclipse.ocl.examples.xtext.completeocl.ui_3.3.0.v20130527-1543.jar	EPL
org.eclipse.ocl.examples.xtext.console_3.3.0.v20130520-1503.jar	EPL
org.eclipse.ocl.examples.xtext.essentialocl.ui_3.3.0.v20130520-1503.jar	EPL
org.eclipse.ocl.examples.xtext.oclinecore.ui_3.3.0.v20130520-1503.jar	EPL
org.eclipse.ocl.examples.xtext.oclstdlib.ui_3.3.0.v20130520-1503.jar	EPL
validation/pivot/ui/ext/	
org.eclipse.ocl.examples.xtext.markup.ui_3.4.0.v20130816-1330.jar	EPL
org.eclipse.ocl.examples.xtext.markup_3.4.0.v20130815-1843.jar	EPL

Table C.2. External libraries utilized by the M4J-DSL Model Generator, divided by (sub-)folder locations.

Library File	**Licence**
org.apache.commons.io_2.0.1.v201105210651.jar	AL 2.0
org.apache.commons.lang_2.6.0.v201205030909.jar	AL 2.0
org.eclipse.emf.common_2.9.2.v20131212-0545.jar	EPL
org.eclipse.emf.ecore_2.9.2.v20131212-0545.jar	EPL
org.eclipse.emf.ecore.xmi_2.9.1.v20131212-0545.jar	EPL
flowdsl/	
com.google.guava_11.0.2.v201303041551.jar	AL 2.0
com.google.inject_3.0.0.v201312141243.jar	AL 2.0
javax.inject_1.0.0.v20091030.jar	AL 2.0
org.antlr.runtime_3.2.0.v201101311130.jar	BSD
org.apache.commons.logging_1.1.1.v201101211721.jar	AL 2.0
org.apache.log4j_1.2.15.v201012070815.jar	AL 2.0
org.eclipse.emf.mwe.core_1.3.1.v201403310351.jar	EPL
org.eclipse.emf.mwe.utils_1.3.2.v201403310351.jar	EPL
org.eclipse.xtext_2.5.4.v201404100756.jar	EPL
org.eclipse.xtext.common.types_2.5.4.v201404100756.jar	EPL
org.eclipse.xtext.util_2.5.4.v201404100756.jar	EPL

Bibliography

[Abbors et al. 2012] F. Abbors, T. Ahmad, D. Truşcan, and I. Porres. MBPeT: A Model-Based Performance Testing Tool. In *VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle*, 2012. (cited on page 7)

[Ashcroft and Manna 1979] E. Ashcroft and Z. Manna. Classics in Software Engineering. chapter The Translation of 'Go to' Programs to 'While' Programs, pages 49–61. Yourdon Press, 1979. ISBN 0-917072-14-6. (cited on page 57)

[Barford and Crovella 1998] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98/PERFORMANCE '98)*, pages 151–160, 1998. (cited on pages 7 and 8)

[Bettini 2013] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd., Aug. 2013. ISBN 978-1-78216-030-4. (cited on page 16)

[Blum 2013] D. Blum. Design of an Application for the Automated Extraction of User Behavior Models from Websites and the Generation of Load Test Scripts, Nov. 2013. URL `http://download.fortiss.org/public/pmwt/theses/Masterarbeit_Blum.pdf`. Master's Thesis, Fakultät für Informatik – Technische Universität München, München, Germany. (cited on page 89)

[b+m Informatik AG 2013] b+m Informatik AG. b+m Informatik AG – Homepage, 2013. URL `http://bmiag.de/`. Last visited November 30, 2013. (cited on pages 3, 16, and 17)

[Brambilla et al. 2012] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012. ISBN 978-1-60845-882-0. (cited on pages 13 and 14)

[Dustin et al. 1999] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, 1999. ISBN 0-201-43287-0. (cited on page 1)

[Fränkel 2013] N. Fränkel. *Learning Vaadin 7*. Packt Publishing Ltd., 2nd edition, Sept. 2013. ISBN 978-1-78216-977-2. (cited on page 22)

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, Mar. 1995. ISBN 0-201-63361-2. (cited on page 54)

[Grönroos 2012] M. Grönroos. *Book of Vaadin: 4th Edition – 1st Revision*. Vaadin Ltd., Nov. 2012. (cited on page 22)

[Grönroos 2014] M. Grönroos. *Book of Vaadin: Vaadin 7 Edition – 2nd Revision*. Vaadin Ltd., May 2014. (cited on page 22)

[JBoss Community 2013] JBoss Community. Hibernate – Homepage, 2013. URL `http://www.hibernate.org/`. Last visited December 5, 2013. (cited on page 17)

[JUnit Team 2014] JUnit Team. JUnit – Homepage, 2014. URL `http://junit.org`. Last visited March 31, 2014. (cited on page 29)

[Kirkup and Frenkel 2006] L. Kirkup and B. Frenkel. *An Introduction to Uncertainty in Measurement using the GUM*. Cambridge University Press, 2006. (cited on page 78)

[Lindholm et al. 2013] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java SE 7 Edition*. Oracle America, Feb. 2013. (cited on page 74)

[Menascé et al. 1999] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A Methodology for Workload Characterization of E-commerce Sites. In *Proceedings of the 1st ACM conference on Electronic commerce (EC '99)*, pages 119–128, 1999. (cited on pages 1, 7, 9, and 80)

[newspaint 2014] newspaint. Creating a Java Sampler for JMeter, May 2014. URL `http://newspaint.wordpress.com/2012/11/28/creating-a-java-sampler-for-jmeter/`. Last visited May 30, 2014. (cited on pages 25 and 27)

[Niemeyer 2014] P. Niemeyer. BeanShell – Simple Java Scripting, version 1.3, 2014. URL `http://www.beanshell.org/manual/bshmanual.html`. Last visited March 30, 2014. (cited on page 25)

[Object Management Group 2012] Object Management Group. Object Constraint Language (OCL), Jan. 2012. URL `http://www.omg.org/spec/OCL/2.3.1`. Last visited March 04, 2014. (cited on page 15)

[Paulson 2005] L. D. Paulson. Building Rich Web Applications with Ajax. In *Computer*, volume 38, pages 14–17. IEEE Computer Society, Oct. 2005. (cited on pages 20 and 21)

[Reimer 2013] S. Reimer. *b+m gear Java 2.9.3 Handbuch*. b+m Informatik AG, Apr. 2013. (cited on pages 16, 17, 18, 63, and 64)

[Schulz 2013] E. Schulz. A Model-Driven Performance Testing Approach for Session-Based Software Systems, Oct. 2013. Student research paper, Kiel University, Kiel, Germany. (cited on pages 1, 2, 12, 31, 32, 33, 77, 78, 86, and 93)

[Schulz et al. 2014] E. Schulz, W. Goerigk, W. Hasselbring, A. van Hoorn, and H. Knoche. Model-Driven Load and Performance Test Engineering in DynaMod. In *Proceedings of the Workshop on Model-based and Model-driven Software Modernization (MMSM '14)*, pages 10–11, 2014. (cited on pages 2, 31, and 32)

[Software Engineering Group – Kiel University 2013] Software Engineering Group – Kiel University. Markov4JMeter – Homepage, 2013. URL `http://se.informatik.uni-kiel.de/markov4jmeter/`. Last visited October 30, 2013. (cited on page 2)

[Stahl and Völter 2005] T. Stahl and M. Völter. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 1st edition, 2005. ISBN 3-89864-310-7. (cited on page 16)

[Steinberg et al. 2009] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2 edition, 2009. ISBN 978-0-321-33188-5. (cited on page 14)

[Subraya and Subrahmanya 2000] B. Subraya and S. Subrahmanya. Object driven Performance Testing of Web Applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software*, pages 17–26, 2000. (cited on pages 7 and 8)

[Taylor et al. 2010] R. N. Taylor, N. Medvidović, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2010. ISBN 978-0-470-16774-8. (cited on page 55)

[The Apache Software Foundation 2013] The Apache Software Foundation. JMeter – Homepage, 2013. URL http://jmeter.apache.org/. Last visited June 24, 2013. (cited on page 1)

[The Apache Software Foundation 2014a] The Apache Software Foundation. JMeter User's Manual, 2014a. URL http://jmeter.apache.org/usermanual/component_reference.html. Last visited March 30, 2014. (cited on page 26)

[The Apache Software Foundation 2014b] The Apache Software Foundation. JMeter Wiki FAQ, 2014b. URL http://wiki.apache.org/jmeter/JMeterFAQ. Last visited May 13, 2014. (cited on pages 21 and 23)

[The Eclipse Foundation 2014a] The Eclipse Foundation. Eclipse Modeling Framework – Homepage, 2014a. URL https://www.eclipse.org/modeling/emf/. Last visited May 04, 2014. (cited on page 14)

[The Eclipse Foundation 2014b] The Eclipse Foundation. OCL/OCLinEcore Eclipsepedia – Homepage, 2014b. URL http://wiki.eclipse.org/OCL/OCLinEcore/. Last visited March 01, 2014. (cited on page 15)

[The Eclipse Foundation 2014c] The Eclipse Foundation. Xtext – Homepage, 2014c. URL https://www.eclipse.org/Xtext/. Last visited April 24, 2014. (cited on page 58)

[The Eclipse Foundation 2014d] The Eclipse Foundation. Graphviz – Homepage, 2014d. URL http://www.graphviz.org/. Last visited May 24, 2014. (cited on page 60)

[The Spring Team 2014] The Spring Team. Spring – Homepage, 2014. URL http://spring.io/. Last visited May 05, 2014. (cited on page 17)

[v. Kistowski et al. 2014] J. v. Kistowski, N. R. Herbst, and S. Kounev. Modeling Variations in Load Intensity over Time. In *Proceedings of the Third International Workshop on Large Scale Testing*, LT '14, pages 1–4, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2762-6. URL http://doi.acm.org/10.1145/2577036.2577037. (cited on pages 89 and 93)

[Vaadin Ltd. 2014] Vaadin Ltd. Vaadin – Homepage, 2014. URL http://vaadin.com/. Last visited May 06, 2014. (cited on pages 18 and 86)

[van Hoorn et al. 2008] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating Probabilistic and Intensity-Varying Workload for Web-Based Software Systems. In *volume 5119 of Lecture Notes in Computer Science*, pages 124–143. Springer, 2008. (cited on pages 2, 8, 36, and 38)

[van Hoorn et al. 2011] A. van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, and N. Wittmüss. DynaMod Project: Dynamic Analysis for Model-Driven Software Modernization. In *1st International Workshop on Model-Driven Software Migration (MDSM), Oldenburg, Germany*, Mar. 2011. (cited on pages 2, 31, 77, and 86) [van Hoorn et al. 2013] A. van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, and N. Wittmüss. DynaMod: Dynamische Analyse für modellgetriebene Software-Modernisierung. Technical Report TR-1305, Department of Computer Science, Kiel University, Germany, Aug. 2013. (cited on pages 2, 77, and 86)

[Wenclawiak et al. 2010] B. W. Wenclawiak, M. Koch, and E. Hadjicostas. *Quality Assurance in Analytical Chemistry: Training and Teaching*. Springer Verlag, 2nd edition, 2010. ISBN 978-3-642-13608-5. (cited on page 80)

[Yang et al. 2007] J. Yang, Z. wei Liao, and F. Liu. The impact of Ajax on network performance. In *The Journal of China Universities of Posts and Telecommunications*. Elsevier B.V., 2007. (cited on page 20)