A Model-Driven
Performance Testing Approach for
Session-Based Software Systems

Student Research Paper

Eike Schulz

October 12, 2013

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Examiner: Prof. Dr. Wilhelm Hasselbring
Advisor: Dipl.-Inform. André van Hoorn (U Stuttgart)
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,

______________________________

ii
The need for high quality software systems, particularly in terms of e-commerce, is undoubted. Customers generally interpret performance characteristics like responsiveness or data throughput as indicators for system quality. Consequently, the overall performance of a system should be kept high. This requirement implies the need for appropriate and reliable performance testing methods. This paper introduces a model-driven performance testing approach, which consists of six tasks, whose underlying ideas are generally applicable to performance tests targeting session-based systems. It provides a systematic procedure for setting up a workload generation environment, based on an analytical model. Furthermore, a model-driven implementation idea for the approach will be presented as a proof-of-concept.

The generation of synthetic workload which complies with the workload generated by a population of real users, is one of the main challenges in performance testing, particularly in load testing. This requires a suitable underlying model for producing appropriate requests to be sent to the considered test system. Our approach uses an analytical model for probabilistic workload generation, which is part of the Markov4JMeter add-on for the performance testing tool JMeter. It is assumed that the behavior specification of the test system provides domain-specific use cases as indicators for the input of the workload generation model, whereas concrete input values are extracted from measurements. The extraction process includes the model-driven part of our approach.

In the evaluation part, a case study system is targeted by the implementation of our approach, demonstrating its practicability. For determining the validity of the approach, several metrics including a methodology for measuring them will be discussed. We will reveal several open issues, aiming to the proof that any results retrieved by using an implementation of our approach are reasonable.
Contents

1 Introduction ........................................... 1
   1.1 Motivation ......................................... 1
   1.2 Document Structure ................................. 2

2 Foundations and Technologies ....................... 3
   2.1 Model-Based Testing ............................... 3
   2.2 Performance Testing .............................. 4
   2.3 Model-Based Performance Testing .................. 5
   2.4 Workload Generation Model based on Markov Chains 7

3 Performance Testing Approach ....................... 13
   3.1 Overview of Approach ............................. 13
   3.2 Manual Specification of Use Cases ............... 15
   3.3 Manual Specification of an Application Model .... 17
   3.4 Template Generation .............................. 18
   3.5 Dynamic Analysis of a Production System ......... 19
   3.6 Automatic Extraction of Behavior Models ......... 21
   3.7 Workload Generation .............................. 23

4 Model-Driven Implementation of the Performance Testing Approach 25
   4.1 Performance Testing with JMeter and Markov4JMeter 25
   4.2 Manual Specification of Use Cases ............... 28
   4.3 Manual Specification of an Application Model .... 29
   4.4 Template Generation .............................. 30
   4.5 Dynamic Analysis of a Production System ......... 31
   4.6 Automatic Extraction of Behavior Models ......... 32
   4.7 Workload Generation .............................. 34

5 Evaluation ............................................ 35
   5.1 AIDA Case Study Setting ......................... 35
   5.2 Qualitative Evaluation ............................ 37
   5.3 Methodology for Quantitative Evaluation .......... 39

6 Conclusions and Outlook ........................... 43
   6.1 Summary ........................................... 43
   6.2 Discussion ........................................ 43
   6.3 Future Work ...................................... 44
List of Figures

2.1 ON/OFF model process .............................................. 7
2.2 Customer Behavior Model Graph for an occasional buyer ........ 8
2.3 EFSM for an e-commerce system .................................. 10
2.4 Sample Application Model illustrating the separation into Session Layer and Protocol Layer ............................................. 12

3.1 Overview of approach .................................................. 14
3.2 Table of use cases and input flows (excerpt) ....................... 16
3.3 Example of a tabular template for a Behavior Model (excerpt) ... 19
3.4 Example of a tabular representation of a Behavior Model (excerpt) ... 19
3.5 Example for a generic format of user session records ............... 20
3.6 Transformation from ABMG to RBMG ................................. 22
3.7 Probabilistic input flow definition for a single Behavior Model (excerpt) .... 24

4.1 JMeter user interface with an example test plan .................. 26
4.2 Input form for a Markov State in Markov4JMeter .................. 28
4.3 Input form for a Markov Session Controller in Markov4JMeter .... 29
4.4 Template generation with Markov4JMeter ........................... 30
4.5 CSV template file generated by Markov4JMeter ................... 31
4.6 Example for a user session records file (excerpt) ................ 31
4.7 EMF class model for Behavior Model extraction ................... 33

5.1 Screenshot of the AIDA-Gear application ............................ 36
Chapter 1

Introduction

1.1 Motivation

Performance has been identified as a business-critical issue for many e-commerce platforms. Menascé et al. [1999] state that “many e-commerce sites, such as those in the financial trading business, have been facing serious problems and financial losses when customers are not allowed to trade in a timely manner”. In many cases, the quality aspect of performance is underestimated. Vokolos and Weyuker [1998] state that “often, the primary problems that projects report after field release are not crashes or incorrect system responses, but rather system performance degradation or problems handling required system throughput”.

Software performance tests can be conducted on distributed systems and on locally installed stand-alone applications as well. However, they are preferably applied to Web-based systems, in particular e-commerce platforms, whose user numbers might increase rapidly, making the system scalability become an important factor to be considered.

In this paper, a model-driven performance testing approach for a given system, shortly referenced to as system-under-test (SUIT), will be introduced. The approach builds on an analytical workload model, which has been introduced by van Hoorn et al. [2008]. The model aims to the generation of probabilistic workload, which emulates the system usage of a certain population of real users. Our approach consists of six single tasks, starting with an analysis of use cases for building an analytical workload model based upon them. The construction of the model includes the manual specification of certain model artifacts, from which a template for probabilistic model parameters can be obtained. These parameters will be extracted from user traces, retrieved through a dynamic analysis of system usage. The extraction of model parameters represents the model-driven part of our approach. The resulting parameterized model is the basis for the workload generation process itself, which is the final task of our approach. The single tasks will be exposed in details, since the underlying idea is generally applicable to performance tests targeting session-based systems. In the evaluation part, a concrete test case will be illustrated.

The performance testing approach introduced in this paper resulted in context of the collaborative research project DynaMod [van Hoorn et al., 2011], which started in the beginning of 2011 and finished in the end of 2012. The evaluation part of this paper considers a Web-based application named AIDA-Gear, which had been provided by the b+m Informatik AG, who were member of the project consortium. This system has been
1. Introduction

targeted by our approach, including an appropriate implementation. For determining the validity of our approach, we will discuss several metrics, which had been identified in the research process. The discussion includes a methodology idea for measuring these metrics. Furthermore, several open issues have been discovered as candidates for future work. We will reveal these issues with the goal of showing that any results retrieved by using an implementation of our approach are reasonable.

1.2 Document Structure

The remainder of this paper is structured as follows:

 ≺ Chapter 2 provides the foundations and technologies for our performance testing approach. First, it gives an introduction to model-based performance testing, illustrating the idea of workload generation and the related ON/OFF model as a fundamental work by Barford and Crovella [1998]. Depending on this work, the analytical model introduced by van Hoorn et al. [2008], which has been used for our performance tests, will be described.

 ≺ Chapter 3 gives an overview of our approach and exposes the single tasks it consists of. The description of the tasks is independent from their implementation, so that the underlying ideas can be easily applied to alternative technical realizations.

 ≺ Chapter 4 presents a proof-of-concept implementation of our approach. It exposes a technical realization using the performance testing tool JMeter and gives an introduction to so-called test plans as well as to the JMeter add-on Markov4JMeter [van Hoorn et al., 2008], which is fundamental for our implementation. The chapter describes the single implementation steps on a certain abstraction level, so that the implementation with alternative techniques can be derived easily from the given realization.

 ≺ Chapter 5 includes the evaluation.

 ≺ Chapter 6 draws the conclusions and outlines future work.
Chapter 2

Foundations and Technologies

The aim of our performance testing approach is the generation of mostly “realistic” request sequences to be sent to a testing system, for measuring characteristics like responsiveness, data throughput or resource utilization. That is, we want to emulate requests, which could originate from a certain population of real users. Therefore, our approach uses an analytical model, which specifies the requests to be sent. The model uses the principles of Markov chains and has been introduced by van Hoorn et al. [2008] as the underlying model of their Markov4JMeter add-on for the performance testing tool JMeter [The Apache Software Foundation, 2013b].

Section 2.1 illustrates the general challenges in model-based testing approaches. Section 2.2 gives some essential definitions for performance testing as well as a classification of performance tests into three different types. Section 2.3 discusses the combination of the model-based testing paradigm and the performance testing paradigm. In Section 2.4, the Markov4JMeter workload generation model which is based on Markov chains will be introduced, including an introduction of its foundations, the so-called Customer Behavior Model Graphs and Extended Finite State Machines.

2.1 Model-Based Testing

The IEEE Computer Society defines (software) testing in their Guide to the Software Engineering Body of Knowledge [Bourque et al., 2004, p. 5–1] as follows:

“Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.

Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.”

The definition aims in its first part particularly to the quality evaluation and the identification of possible defects for improving a product, considered as SUT. It continues with a more detailed specification of software testing with focus on dynamic verification. Utting and Legeard [2007, p. 3] associate dynamic verification with the execution of a program, using specific input values and aiming to the detection of failures in the program’s behavior. They state that in contrast, the execution of a program is not required for static verification
2. Foundations and Technologies

techniques. In those cases, the tester has insight to the program code and can analyze it in terms of correctness.

The primary intention of model-based testing (MBT) is the automatic generation of test suites as well as the automation of test activities for a given system. The approach is based upon a data model which specifies input values for the system to be tested. Consequently, the data model needs to be updated, in case any changes to the system regarding its input are made. Dalal et al. [1999] emphasize on the other hand the effectivity of test generation for frequently changed systems, for which tests can be generated rapidly with the use of an updated model instead of writing time-consuming and error-prone tests by hand. The size of a data model is not arbitrary, because it must be small in relation to the SUT for keeping its production costs low, but it must contain enough details for an exact description of the characteristics being tested [Utting and Legeard, 2007].

Tretmans and Brinksma [2003] remark that test automation is ambiguous through traditionally referring to the automation of test execution, and sometimes to the automation of test analysis.

2.2 Performance Testing

Abbors et al. [2012] define performance testing as follows:

“The idea behind performance testing is to validate the system under test in terms of its responsiveness, stability, and resource utilization when the system is put under certain synthetic workload in a controlled environment.”

Thereby, synthetic workload denotes the workload which is generated by corresponding tools, shortly referenced to as workload generators or load generators. Menascé [2002] states that a load generator emulates the behavior of real users by continuously submitting requests to the SUT and waiting for periods of time (think times) in between. It can emulate an unlimited number of concurrent users, each of them referred to as virtual user. Particularly in the context of testing web sites, the sessions concept is important. A session is defined as “a sequence of requests of different types made by a single user during a single visit to a site” [Menascé et al., 1999]. The idea can even be applied to stand-alone applications, so that it does not restrict the type of systems to be tested.

In context of workload generation for Web-based server systems, Krishnamurthy et al. [2006] distinguish between two steps, namely trace generation and request generation. In the trace generation step, the traces of requests to be sent to the SUT are created. These traces indicate the sequences of requests to be submitted to the SUT in the request generation step.
2.3. Model-Based Performance Testing

Software performance tests can be classified into different types. Subraya and Subrahmanya [2000] distinguish between the following three test types:

- In a **load test**, the synthetic workload corresponds to the workload of users in the real world, including think time delays. Load testing aims to the observation of the SUT under day to day conditions [Subraya and Subrahmanya, 2000].

- In a **stress test**, the synthetic workload corresponds to the workload of a load test without any think time delays [Subraya and Subrahmanya, 2000]. Stress testing aims to the observation of the SUT in a state of extreme load for so-called **bottlenecks** being identified [Draheim et al., 2006]. That is, extreme synthetic workload is put to the SUT for detecting components which limit the upper system capacity.

- An **endurance test** is a long duration load or stress test with a much longer execution period. Endurance testing targets applications of high importance, which are intended to run for a long time [Subraya and Subrahmanya, 2000].

Since the underlying ideas of the test variations are very similar to the essential load test approaches, the remainder of this paper will focus on load testing. Menascé [2002] states that a load test is valid only if the behavior characteristics of virtual users and real users are similar to each other and that therefore, the following requirements must be met:

- virtual users follow behavioral patterns which are similar to those of real users,
- virtual users simulate realistic think times between their requests,
- virtual users leave a session, in case the system response time exceeds a certain limit.

Hence, Abbors et al. [2012] describe the generation of workload as one of the main challenges in performance testing, since, for example, important user types with a significant impact on the performance of the system might not be identified.

2.3 Model-Based Performance Testing

This section discusses the idea of model-based performance testing, whereby the workload generation of a performance test is the essential part to be discussed. Section 2.3.1 illustrates the distinction between **trace-based** and **analytical** workload generation and points out the idea of model-based workload generation. Section 2.3.2 describes the **ON/OFF Model** by Barford and Crovella [1998] as an example for an analytical model that can be used for generating model-based workload.
2. Foundations and Technologies

2.3.1 Model-Based Workload Generation

Barford and Crovella [1998] distinguish between two approaches for reproducing typical workload, which mimics a population of real users: trace-based and analytical workload generation. Trace-based workload generation uses prerecorded records of the past and thereby mimics activity of a known system. Its implementation is easy, but it treats the workload itself as a “black box” and therewith gives no insight into causes of system behavior. Analytical workload generation uses mathematical models for various workload characteristics and generates corresponding output. The challenge with analytical models is the identification and empirical measurement of the characteristics, as well as their combination to a single output workload [Barford and Crovella, 1998].

Analytical workload generation for performance tests combines the MBT paradigm and the previously discussed performance testing paradigm. Hence, approaches of that kind are shortly referenced to as model-based performance tests. A workload generation model, which can be used for model-based performance tests, will be described next.

2.3.2 ON/OFF Model

The ON/OFF workload generation model introduced by Barford and Crovella [1998] is an analytical-based approach for imitating closely a stream of requests originating from a fixed population of users. Each user’s request stream is defined as a single process in an endless loop. The main characteristics of the model is the alternation of such a process between making requests (ON times) and lying idle (OFF times). Thereby, two kinds of OFF time are distinguished:

- Inactive OFF time corresponds to the time between transfers of objects. It results from the fact that a user generally needs some time for reaching a decision before he makes a new request to the system. In that time, which complies with the think time discussed in Section 2.2, the user is in a state of “inactivity” in terms of making any requests to the system.

- Active OFF time corresponds to the time between transfer of components of a single object, e.g., images of a requested Web-site. It includes wait times for tasks to be finished, e.g., file transfers or database respond delays. The user is forced to wait for his next request, rather than continually making one request just after another. In that time, the user is in a state of “activity” in terms of making any requests to the system, but the requests have no effect.

Figure 2.1 shows an example: It depicts the time bar of a process for a user who requests a Web-page with two embedded items. The first point in time, labeled User Requests Page, denotes the user request for the Web-page. The requested page is being transferred in the ON Object period: In the URL 1 period, the textual content of the Web-page is being transferred. The process runs in an idle period afterwards, caused by a system delay; this
is an Active OFF period, since any user requests have no effect. The URL 2 and URL 3 periods for the embedded items to be transferred correspond the same. After the Web-site has been transferred, the process runs in an Inactive OFF period, since the user needs think time before sending a new request.

![Diagram: ON/OFF model process](image)

**Figure 2.1.** ON/OFF model process [Barford and Crovella, 1998]

The model has been used by Barford and Crovella in their workload generation tool SURGE (Scalable URL Reference Generator) [Barford and Crovella, 1998]. Furthermore, the authors showed that additional dependencies between requests lets the workload increase in contrast to workload which is generated by making independent requests.

Other model-based workload generation models will be discussed in the next section.

## 2.4 Workload Generation Model based on Markov Chains

The workload generation model described in this section allows the definition of probabilistic user behavior. It is based upon a special variant of Markov chains called Customer Behavior Model Graphs which were introduced by Menascé et al. [1999]. It has been used in the Markov4JMeter add-on [van Hoorn et al., 2008] for the performance testing tool JMeter [The Apache Software Foundation, 2013b]. For the description of allowed user input sequences to be applied to session-based systems, the model additionally uses the concept of Extended Finite State Machines, which were introduced by Shams et al. [2006] according to the work of Krishnamurthy et al. [2006].

To begin with, the fundamental concepts will be discussed in the Sections 2.4.1 and 2.4.2, before the workload generation model itself will be described in Section 2.4.3.

### 2.4.1 Customer Behavior Model Graphs

Menascé et al. [1999] discovered that in e-commerce applications, different navigational patterns can be observed for different groups of customers, each indicating a certain user type. For example, two user types in terms of a Web-store might be occasional and heavy
2. Foundations and Technologies

buyers. The behavior of a certain user type can be described by using a Customer Behavior Model Graph (CBMG), which is a state transition graph consisting of a set of nodes for possible states (e.g., Search, Add to Cart, and Pay) and transitions between the states. Each transition is labeled with a probability value, which specifies the percentage probability of going from a state to the next. In short, a CBMG is a special variant of Markov chains which characterizes a certain user type by the means of its transition probabilities.

Figure 2.2 shows an example for a CBMG which models the behavior of an occasional buyer. In particular, the transition probability of 0.05 from state Select to state Add to Cart indicates a relatively low purchasing power of the modeled user type.

Figure 2.2. Customer Behavior Model Graph for an occasional buyer [Menascé et al., 1999]

Menascé et al. [1999] emphasize that a CBMG describes the behavior of users from the viewpoint of the server, based only on requests which arrive on server side. User requests which can be resolved through caching on client or proxy side are not described by the CBMG. In particular, the CBMG does not include the corresponding transitions. This is no restriction for the workload characterization, since those requests do not affect any server resources [Menascé et al., 1999].

2.4.2 Extended Finite State Machines

Krishnamurthy et al. [2006] discovered that trace-based evaluation of session data can result in wrong models and demonstrated this with the following example: Consider the valid Web-store sessions [Home, View, Add, View, Add, Delete, Purchase] and [Home, View, Add, Purchase]. The first session describes a shopper who adds two products to the shopping cart and then deletes one of the products from the shopping cart before purchasing the
other product. The second shopper adds a product to the shopping cart and purchases that product. Since only the first session contains a request for deletion, one resulting conclusion would be that after each Delete follows a Purchase request. That is, the dependency between Purchase and the two Add requests has not been recognized. This (wrong) assumption allows also sessions like [Home, View, Add, Delete, Purchase], leading to a Purchase in spite of an empty shopping cart.

As a solution, Shams et al. [2006] suggest to use Extended Finite State Machines (EFSMs), to ensure that certain states can only be reached under predetermined conditions. They describe an EFSM as a structure consisting of states, input symbols, output symbols, variables, and transitions between the states. In contrast to a Finite State Machine (FSM), an EFSM additionally contains predicates and actions defined on the specified set of variables. Referring to the paper of Shams et al. [2006], an EFSM can be described as a tuple

\[(\Sigma, \Gamma, S, \vec{x}, T)\]

with \(\Sigma, \Gamma, S, \vec{x}, T\) being finite sets of input symbols, output symbols, states, variables respectively transitions. A transition \(t \in T\) is defined as a tuple

\[(s, s', a, o, P, A)\]

with \(s, s'\) being the current state and the next state. The symbols \(a, o\) denote input and output. \(P(\vec{x})\) represents a predicate, and \(A(\vec{x})\) represents an action, both defined on the current variable values. The machine is initialized in a state \(s_{initial} \in S\) with initial variable values \(\vec{x}_{initial}\), and it works as follows: if the machine is currently in state \(s \in S\) with variable values \(\vec{x}\) and it receives an input \(a\), the EFSM fires a transition \((s, s', a, o, P, A)\), if the predicate \(P(\vec{x})\) evaluates to true. In that case, the machine produces the output \(o\), the variable values are modified as per the action \(A(\vec{x})\), and the EFSM moves to state \(s'\).

Figure 2.3 shows an EFSM for an e-commerce system, with states \(s_0, \ldots, s_8\), input symbols corresponding to the possible requests being made to the system (Home, Browse, Sign in, \ldots), and two variables signed_on and items_in_cart of type boolean respectively integer. The predicates and actions are assigned to their regarding transitions, indicated by characters \(P\) and \(A\), respectively. The EFSM models the constraints that a user needs to be signed on before making a purchase. Furthermore, items can only be deleted from cart, if the cart is not empty. In the depicted EFSM, a Purchase in spite of an empty shopping cart as discussed before is still possible, but an appropriate constraint can be added easily.

### 2.4.3 Workload Generation Model in Markov4JMeter

The workload generation model used in the Markov4JMeter add-on for JMeter consists of the following components [van Hoorn et al., 2008]:

- An Application Model, which specifies valid sequences of service invocations within a user session. Additionally, it contains the protocol-related information which is required for the generation of valid requests. The Application Model itself is specified as a
2. Foundations and Technologies

![EFSM for an e-commerce system](Shams et al., 2006)

**Figure 2.3.** EFSM for an e-commerce system [Shams et al., 2006]

A hierarchical finite state machine, based on the EFSMs as they have been introduced before.

- A set of **User Behavior Models**, each of them structured as a Markov chain according to the Application Model. A Behavior Model can be associated with an individual user type.

- A **User Behavior Mix**, which specifies the relative frequency of each Behavior Model to occur in a workload generation process.

- A **workload intensity**, which is defined as the number of virtual users in a workload.
2.4. Workload Generation Model based on Markov Chains

generation process. Thereby, the number of users might vary.

The workload intensity is described as a function which defines the number of users at a
certain point in time of a workload generation process. Since we conduct our performance
tests without a time-dependent varying number of users, the main focus of the following
discussion will be put on the Application Model, the User Behavior Models, and the User
Behavior Mix. All three components will be described in the remainder of this section,
regarding to the definitions given by van Hoorn et al. [2008]. They define an Application
Model as follows:

“An Application Model is a two-layered hierarchical finite state machine. It consists
of a Session Layer modeling the valid sequences of service invocations within a user
session and a Protocol Layer specifying the related protocol details.”

The Session Layer consists of an EFSM defined on a set of states, corresponding to the
services provided by the application. A transition between two states denotes a valid
sequence of service invocations within a session. Therefore, a transition can be labeled with
guards and actions optionally, whereas a guard is a boolean expression which defines the
condition under which the related transition fires. That is, the target state of the transition
becomes the current state of the EFSM, in case the guard condition is fulfilled. An action
defines the list of statements to be executed, in case the related transition fires [van Hoorn
et al., 2008].

The Protocol Layer contains for each state of the Application Model an associated EFSM.
In contrast to the Session Layer, the EFSMs of the Protocol Layer define the protocol-specific
request sequences to be invoked for a service request. Their transitions can be labeled with
guards and actions, too [van Hoorn et al., 2008].

An example of an Application Model is shown in Figure 2.4, illustrating the separation
into a Session Layer and a Protocol Layer: The Session Layer contains an EFSM with three
states S0, S1, and S2. The transitions define the valid sequences passages from one state to
another. For example, the transition from S2 to S0 is labeled [b\neq 0]/b=1, consisting of the
guard \(b\neq 0\) and the action \(b=1\). The guard \(\neq 0\) declares that the related transition might
only be fired, in case the variable \(b\) does not equal 0. The action \(b=1\) declares that the value
1 is assigned to \(b\), if the transition fires.

The EFSM of the Protocol Layer which is assigned to state S0 of the Session Layer
depicted in Figure 2.4, is defined on the three protocol states \(a.shtml\), \(b.shtml\), and \(c.shtml\)
which correspond to URIs for HTTP requests. After the request for \(a.shtml\) has been
received, the guard \(a > 0\) is evaluated. Its result is 0 or 1, indicating the next state to be
taken.

A User Behavior Model is defined by van Hoorn et al. [2008] as follows:

“A User Behavior Model constitutes a probabilistic model of service invocation sequences
within simulated user sessions. [...] A class of similarly behaving users can be
represented by a single Behavior Model.”
Formally, a User Behavior Model $B_{A,i}$ for an application $A$ is defined as a tuple

$$ (S \cup \{\$\}, P, z_0, f_{tt}) $$

with $S$ being the set of states on which the related Markov chain is defined, including an initial state $z_0 \in S$. The state $\$ is a dedicated exit state. $P$ is an $n \times n$-matrix for $n = |S|$, denoted as $P = [p_{i,j}]$. A matrix entry $p_{i,j}$ specifies the probability for a transition from state $i$ to state $j$ being fired, that is the service associated with state $s_j$ being invoked in state $s_i$ by the user. The distribution function $f_{tt}$ specifies the think time. In the implementation of our approach, the think times are modeled by using random values in JMeter as described in Section 5.2.6, hence we will not go into further details about distributions at this point.

The User Behavior Mix for an application $A$ is defined as a set

$$ \{(B_{A,0}, p_0), \ldots, (B_{A,n-1}, p_{n-1})\} $$

which assigns a relative frequency $p_i$ to each User Behavior Model $B_{A,i}$. A tuple $(B_{A,i}, p_i)$ indicates that the session input of a virtual user, which corresponds to the User Behavior Model $B_{A,i}$, occurs with a relative frequency of $p_i \in [0,1]$ during workload generation. That is, each $p_i$ denotes a percentage value, and the sum of all values must be 1, according to 100 percent.

In the workload generation process, the model is used as follows: The User Behavior Mix determines the user type to be emulated next by selecting the corresponding User Behavior Model with regards to the assigned relative frequencies. In the selected User Behavior Model, a sequence of services is invoked according to the probabilities specified in the related Markov chain.
Performance Testing Approach

In this chapter, our approach for testing the performance of a given SUT will be described. We distinguish between a production system (PS) and a system-under-test (SUT). This results from the aim of the DynaMod project [van Hoorn et al., 2011], where our approach resulted from. The aim of the project was the semi-automatic transformation of a legacy system into a modernized system. As a part of the process, a prototype of a modernized system has been created for being targeted by our performance tests. This system is identified hereinafter as SUT, that is, a system being monitored in terms of its performance characteristics like responsiveness or stability. The other system is identified as PS, that is, a system which has been already in practical use and from which concrete usage records exist. It needs to be pointed out that the approach even works for PS and SUT being the same. For a consistent view on both systems, the functionality of both systems is being considered on use case level for a mostly abstract view. This idea will be discussed more detailed in the following description of the approach and can be generalized to the case that both systems are implemented as a unique instance.

Section 3.1 gives an overview of the approach, which comprises six tasks to be done. These tasks will be discussed in the Sections 3.2 – 3.7, with each task being described in an own section.

3.1 Overview of Approach

This section gives an overview of our approach. The approach includes six tasks for retrieving certain artifacts as results. Figure 3.1 shows the tasks and artifacts of the approach as well as their relations between each other. Solid arrows denote tasks, that is, transitions from artifacts to others, while dashed arrows denote dependencies. Each transition is signed with a label in a filled oval shape. A label signifies the activities to be done for retrieving the result artifact targeted by the related transition. A gray-colored label indicates that the activities of the related task are out of scope for this thesis. In contrast, a black-colored label indicates full insight into the activities of the related task. Each label number refers to a section, which describes the related task in details. For a task, additional source input provided by an artifact might be required. This kind of dependency is depicted as a dashed arrow leading from an input-providing artifact to the transition label of a requiring task.
3. Performance Testing Approach

Figure 3.1. Overview of approach, based on van Hoorn et al. [2011]
The remainder of this chapter clarifies the idea of the single tasks to be performed for applying the approach. The section numbers correspond to the numbers depicted above the transition labels in Figure 3.1.

The approach starts with an analysis of use cases. It is assumed that the behavior specification of a considered system provides a set of domain-specific use cases. In our approach, the specification of the PS provides use cases, which define the functional behavior of the PS and the SUT as well. However, they do not necessarily need to be implemented in the SUT yet. The use cases whose functionality is already provided by both systems, are the only ones to be considered. For them, an analysis of user input for a use case execution is done for each system. This will be described in Section 3.2.

The result of this task is a set of common use cases, which has significant impact on the manual specification of an Application Model and for the automatic extraction of Behavior Models from user session records as well. For a uniform treatment of both systems, the abstraction level in our approach implies that the states of these models correspond with implemented use cases. Section 3.3 describes the manual specification of the Application Model. The Application Model provides a template for storing the values of Behavior Models, in particular the transition probabilities. Section 3.4 illustrates the generation of such a template. The values to be filled in are gained from an automatic extraction of Behavior Models, depending on user session records. These records can be obtained through a dynamic analysis of the PS, as described in Section 3.5. The extraction of the Behavior Models and the Behavior Mix from these session records is discussed in Section 3.6. Finally, Section 3.7 illustrates the workload generation process, which depends on the used workload generation model.

### 3.2 Manual Specification of Use Cases

As noted before, we distinguish between a PS and a SUT, whereas the SUT is a prototype of a modernized PS version. Consequently, the SUT does not provide full functionality of the PS. For our performance tests, respectively their evaluation, we need to consider a uniform set of services provided by both systems. This set indicates a common Application Model as described in Section 2.4.3, with each state of the included Session Layer graph representing a provided service of the PS and the SUT as well. As a part of the workload generation model, the Application Model defines the valid sequences of user input for both systems. In contrast, both systems have individual input mechanisms, so we need to increase the abstraction level for a uniform handling of input. Hence, the provided services of both systems are considered on use case level. Therewith, each use case can be associated with a corresponding state of the Session Layer graph in the Application Model.

In general, a system behavior specification contains a set of use cases, which define the domain-specific services to be provided of the related system. Those use case might be grouped into several packages, according to their functional scopes. For the sake of clarity, we divide use cases into packages in our examples, though it is actually not necessary for
3. Performance Testing Approach

the further tasks of our approach.

For a suitable utilization, the given use cases need to be analyzed in terms of their realization within the SUT, assuming that the PS provides the specified functionality anyway. Additionally, an investigation of the input flows to be provided by a user for executing certain use cases is necessary for retrieving appropriate input sequences to be emulated. The result is a table of use cases and input flows, which contains the base information needed for constructing the input sequences of our performance tests. Figure 3.2 shows an excerpt of the table, depicting only its first rows.

<table>
<thead>
<tr>
<th>Pkg</th>
<th>UC#</th>
<th>Use Case Name in PS</th>
<th>PS – Input Flow</th>
<th>In SUT</th>
<th>SUT – Input Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>登录</td>
<td>1. click “Login” button 2. input data + click “OK” button</td>
<td>yes</td>
<td>1. click “Login” button 2. input data + click “OK” button 3. close frame</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>添加项</td>
<td>1. click “Add” button 2. input data + click “Save” button</td>
<td>yes</td>
<td>1. click “Add” button 2. input data + click “Save” button 3. close frame</td>
</tr>
<tr>
<td>02</td>
<td></td>
<td>edit item</td>
<td>1. click “Edit” button 2. select item (becomes editable) 3. input data + click “Save” button</td>
<td>yes</td>
<td>1. click “Edit” button 2. select item (becomes editable) 3. input data + click “Save” button 4. close frame</td>
</tr>
<tr>
<td>05</td>
<td></td>
<td>删除项</td>
<td>1. click “Delete” button 2. select item (will be deleted)</td>
<td>yes</td>
<td>1. click “Delete” button 2. select item (will be deleted) 3. close frame</td>
</tr>
<tr>
<td>04</td>
<td></td>
<td>导出项 (DB)</td>
<td>1. menu: “File” → “Export Items” 2. input filename + click “OK” button</td>
<td>no</td>
<td>1. menu: “File” → “Export Items” 2. input filename + click “OK” button</td>
</tr>
<tr>
<td>05</td>
<td></td>
<td>导入项 (DB)</td>
<td>1. menu: “File” → “Import Items” 2. input filename + click “OK” button</td>
<td>no</td>
<td>1. menu: “File” → “Import Items” 2. input filename + click “OK” button</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>添加分类</td>
<td>1. click “Add” button 2. input data + click “Save” button</td>
<td>yes</td>
<td>1. click “Add” button 2. input data + click “Save” button 3. close frame</td>
</tr>
<tr>
<td>02</td>
<td></td>
<td>edit classification</td>
<td>1. click on “Edit” button</td>
<td>1. click “Edit” button</td>
<td>1. click “Edit” button</td>
</tr>
</tbody>
</table>

Figure 3.2. Table of use cases and input flows (excerpt)

The table contains for each use case the related input flow on the PS respectively the SUT. Its header columns from left to right are as follows:

- **Pkg**: Identification number of the package to which the related use case is assigned.
- **UC#**: Identification number of the use case within the package.
- **Use Case Name**: Name of the use case as laid down in the PS specification.
- **in PS**: Boolean value indicating whether the PS supports the use case’s functionality.
- **PS – Input Flow**: Sequence of activities to be performed by a user who executes the use case’s service on the PS.
- **in SUT**: Boolean value indicating whether the SUT supports the use case’s functionality.
3.3 Manual Specification of an Application Model

$\Rightarrow$ SUT - Input Flow: Sequence of activities to be performed by a user who executes the use case’s service on the SUT. Since different variants of activity sequences may exist for service execution, a use case possibly indicates different variants of input flow.

For use cases, whose functionality is not provided by the SUT, related input flow does not exist. For example, the table rows for the use cases named export items (DB) and import items (DB) in Figure 3.2 denote use cases, which are only implemented in the PS. Hence, the related cells for the SUT input flows are left blank. In contrast, the use cases whose functionality is implemented in the SUT, indicate service-related input flow, possibly in different variants, which can be emulated through synthetic workload by generating appropriate requests. These use cases correspond to the yes entries in the in SUT table column, and they determine the states of the Session Layer graph to be defined for the Application Model. This will be described in Section 3.3. The use cases have significant impact on the automatic extraction of the Behavior Models, too. This will be discussed in Section 3.6.

3.3 Manual Specification of an Application Model

As a part of the workload generation model described in Section 2.4.3, an Application Model defines the valid sequences of service executions as well as the protocol-level requests to be invoked when a service is executed.

The valid service execution sequences are defined by the state machine of the Session Layer, with states that correspond to the available system services. In our approach, these services can be associated with the domain-specific use cases, whose functionality is implemented in the SUT. The identification of them has been discussed in Section 3.2. Consequently, the introduced table of use cases and input flows, shown by example in Figure 3.2, gives us the states of the Session Layer graph as exactly those use cases, whose functionality is provided by the SUT. In Figure 3.1, this dependency is indicated by the dashed arrow leading from the use case diagram to the manual specification task of the Application Model. As a result, the states of the Session Layer graph as well as the valid transitions between them can be obtained. The validity of service execution sequences is domain-specific, so the use cases must be analyzed accordingly. For example, certain services might only be executed, if a user has signed in before.

The Protocol Layer depends on individual SUT characteristics, hence a general procedure for building the related state machines cannot be given. If the code of the SUT is known, it can be analyzed for obtaining the necessary protocol information. If the SUT is a “black box”, it can be only analyzed in terms of its behavior specification. The SUT defines with its navigational patterns certain sequences, which can be indicators for behavioral issues, for example, login forms as conditions for further input. In case of Web-Based systems, a proxy server might help for retrieving protocol information. A concrete example will be demonstrated in Chapter 5. In Figure 3.1, the dashed arrow leading from the SUT to the manual specification task of the Application Model indicates this dependency, too.
3. Performance Testing Approach

Besides the Application Model including the Session Layer and Protocol Layers, the workload generation model additionally contains a set of one or more Behavior Models and a Behavior Mix. In our approach, an Application Model provides a template for the Behavior Models for being filled with concrete probability values regarding to Markov chains as described in Section 2.4.3. The template generation will be discussed in Section 3.4 as a further task of our approach. The actual probability values for the Behavior Models result from an automatic extraction of session-based usage data, which will be discussed in Section 3.6.

3.4 Template Generation

Behavior models represent the usage characteristics of a system and can be gained from related user session records, which need to be determined in the context of a dynamic system analysis. As a result, the probability values $p_{ij}$ of matrix $P$ can be extracted from these records. A tabular template for storing the extracted values can be derived from the related Application Model. This section introduces a template for storing Behavior Models. Each filled instance of the template represents a related Behavior Model with a common set of states, a dedicated exit state, an initial state, and the probability values of transitions between the states.

In Section 2.4.3, a Behavior Model for an application $A$ has been formalized as a tuple $(S \cup \{\$\}, P, z_{0}, f_{u})$ with a set $S$ of states, an exit state $\$$, an initial state $z_{0}$, and a matrix $P = [p_{ij}]$ of transition probabilities between the states. The Session Layer of an Application Model defines the states to be contained in set $S$ of a Behavior Model. These states, which are associated with use cases in our approach, indicate a table with the use cases being ordered as illustrated by the example in Figure 3.3: the table headers contain all use cases listed in top-bottom respectively left-right order. The initial state, even associated with a use case, is marked with an asterisk, and a dedicated exit state $\$$ has been added. Some columns in between have been left out for the sake of more clarity, indicated by dots.

Through the automatic extraction of Behavior Models from session records, this tabular template $T$ can be filled with probability values, whereas a table entry $t_{ij}$ specifies the probability for use case $j$ being executed after use case $i$. Consequently, the column for $\$$ does not need a corresponding row, as the exit state has no outgoing transitions to any executable use case. Each Behavior Model of the workload model has individual probability values, so a related instance of the template needs to be filled with individual values for each model. The Session Layer of an Application Model provides validity information for use case execution sequences. If use case $j$ is not allowed to be executed after an execution of use case $i$, the template entry $t_{ij}$ might be predefined with 0.

Since the probabilities of all outgoing transitions of a (Markov) state $s_{i}$ must result in a sum of 1, the value of $\$$ in row $i$, which is $t_{i,n+1}$ for $n := |S|$, must match the constraint

$$t_{i,n+1} = 1 - \sum_{j \in S} t_{ij}.$$
3.5 Dynamic Analysis of a Production System

As stated before, a dynamic analysis needs to be conducted for retrieving user session records, which can be used for the automatic extraction of Behavior Models. Section 3.5 illustrates the procedure of such an analysis in more details. Section 3.6 discusses the automatic extraction of transition probabilities from the gained session records as a further task of our approach.

3.5 Dynamic Analysis of a Production System

In context of the DynaMod project [van Hoorn et al., 2011], a dynamic analysis has been conducted on the PS for retrieving user session records. Though the dynamic analysis of systems is out of scope for this thesis, we give an outline of a generic trace-based procedure, targeting systems, which contain an event-based user interface. The idea depends on the construction of use case related transition models for being used as acceptors for use cases, which are executed by a user. They can be used for identifying use cases in a given amount of user traces.

As a step of preparation, the considered system needs to be instrumented for executing a given set of use cases in a controlled environment. An appropriate instrumentation of the system contains techniques for recording all input events of the user interface. The records might even contain the event sources, that is information about the input forms which fired the related events. Any redundant information might be removed later by using dedicated filtering methods. The controlled execution of all use cases gives the event sequences for each use case. From these event sequences, a transition model can be constructed for being
used as an acceptor for the related use case execution.

The main task consists of the identification of use cases in a given amount of user traces. This can be achieved by reducing the traces to sequences of input events for applying the transition models retrieved in the preparation step. Therefore, the user traces need to be split to session records at first. Any filtering methods of the preparation step might be applied to those session records as well, resulting in sequences of input events without redundant information. The transition models can now be applied on these sequences for the identification of use cases. The reconstruction of sessions can be done by using certain threshold values for time intervals between records. For example, two records might be combined to a single session, in case their time interval is less than two hours.

Formally, the dynamic analysis provides a set $S = \{S_1, S_2, \ldots, S_n\}$ of session records, whereas a single record $S_i$ is a finite sequence of use case executions $s_{i1}, s_{i2}, \ldots, s_{in}$ with $n_i \geq 1$ being the number of use case executions in the related session. Let $t_{\text{start}}^i$ be the start time and $t_{\text{end}}^i$ the end time of use case execution $s_{ik}$ in session record $S_i$. Time information can be included to $S$ by extending its elements to tuples $(s_{ik}, t_{\text{start}}^i, t_{\text{end}}^i)$.

Each session record consists of at least one executed use case, so the start time of $S_i$ is given by $t_{\text{start}}^1$, and the end time is $t_{\text{end}}^n$. The session records can be stored sequentially in a generic format, which is depicted in Figure 3.5 by example. The leading number $id_i$ of a line is an additional identifier for session record $S_i$. The remaining elements of each line represent the session information as discussed before.

![Figure 3.5. Example for a generic format of user session records](image)

The resulting records can be used for the automatic extraction of usage characteristics, Behavior Models in particular. This task will be described in Section 3.6.
3.6 Automatic Extraction of Behavior Models

This section exposes the automatic extraction of Behavior Models from user session records as an additional task of our approach. A session has been defined in Section 2.2 as “a sequence of requests of different types made by a single user during a visit to a site” [Menascé et al., 1999]. The requests of a session indicate service invocations, which can be associated with use cases, regarding to the abstraction level of our approach. This has been discussed in Section 3.2. Consequently, a session can be associated with a sequence of use case executions. Section 3.5 discussed the dynamic analysis of a system for obtaining appropriate user session records. These records provide usage characteristics, which indicate different user types and their related Behavior Models, in particular. With the use of clustering methods as introduced by Menascé et al. [1999], similar user behavior characteristics can be identified. However, it is out of scope for this paper. The main challenge in extracting Behavior Models is the determination of the transition probabilities, which will be discussed hereinafter. If time information is provided by the session records, think times can be extracted, too. Those issues will be discussed in the remainder of this section.

For each Behavior Model, the set of its states including initial and exit states must be determined, as well as the transitions between the states with their probabilities in particular. Furthermore, the think times must be defined. The set of states is a subset of the considered use cases, since we associate states with use cases. A session record is given as a sequence of use case executions. Consequently, transitions occur within a session between an executed use case and its succeeding executed use case. The executed use cases of a session correspond to the states of the related Behavior Model, including a dedicated exit state. For obtaining a Behavior Model from a session record, we proceed in three steps:

1. Determination of transitions between use case executions and their successors, and counting their occurrences.

As described above, the given set of use case executions within a session and the determined transitions between them indicate a graph with use cases as states and transitions between them. Labeling these transitions with their count values, indicates a graph structure that we denote as Absolute Behavior Model Graph (ABMG). It can be formalized as a tuple

\[(S \cup \{\$\}, A, z_0)\]

with \(S\), $, and \(z_0\) as for Behavior Models and \(A = [a_{ij}]\) as a matrix, analogous to the definition of \(P\) in a Behavior Model, but containing absolute values.

2. Transformation of the ABMG to a graph with probability transition values, derived from the absolute values of the ABMG. We denote the resulting graph as Relative Behavior Model Graph (RBMG). It can be formalized as

\[(S \cup \{\$\}, P, z_0)\]
3. Performance Testing Approach

with $S$, $\$, $z_0$, and $P$ as for Behavior Models. That is, an RBMG can be interpreted as a Behavior Model without think times.

3. Extraction of think times from the session records and specification of the distribution function $f_{tt}$. For this step, time information needs to be provided by the records.

In step 1, an ABMG $(S \cup \{\}\), $A$, $z_0$) for a session record can be constructed by scanning the record straightforward. Each execution indicates a unique representive use case to be added to $S$. The first executed use case indicates the start state, and an exit state $\$ needs to be added. The $n \times n$-matrix $A$ for $n = |S \cup \{\}|$ is initialized with zeros. The session record can now be scanned for succeeding use case executions. A succeeding execution determines the target state of a transition, with its predecessor being the source state. For each occurrence of a transition from state $s_i$ to state $s_j$, its related matrix value $a_{ij}$ is increased by 1. The last use case execution of a session indicates a transition from the related state to the exit state.

In step 2, the absolute transition values of a given ABMG $(S \cup \{\}, A, z_0)$ must be converted to relative values for transforming the ABMG to an RBMG. For a state $s_i$, the relative value $p_{ij}$ of a transition to state $s_j$ is defined as

$$ p_{ij} := \frac{a_{ij}}{\sum_{k=1}^{n} a_{ik}} \text{ with } n = |(S \cup \{\}|. $$

Figure 3.6 shows an example of an ABMG and its resulting RBMG. The ABMG consists of three states $s_1$, $s_2$, $s_3$ and an exit state $\$. Each transition is labeled with an absolute value $a_{ij}$, indicating the number of transition occurrences from state $s_i$ to state $s_j$. The conversion from absolute values to relative transition values leads to the RBMG depicted on the right side in Figure 3.6: For example, the sum of all values assigned to outgoing transitions of $s_1$ in the ABMG is $2 + 6 = 8$, implicating relative values $\frac{2}{8} = 0.25$ and $\frac{6}{8} = 0.75$, assigned to the related transitions of the RBMG.

Figure 3.6. Transformation from ABMG to RBMG. The states $s_1$, $s_2$, and $s_3$ can be associated with the execution of use cases, for example 1.01 add item, 1.02 edit item respectively 1.03 delete item.

In step 3, the user think times are determined. Think times of real users can only be determined, if time information for use case executions is provided by the session records.
3.7 Workload Generation

In case time borders for each use case execution are given, the average think time \( tt_i \) of a user in state \( s_i \) can be calculated, indicating a think time distribution \( f_{ti} \). This step has not been done in our approach. The think times we use are based on estimated values, which will be discussed later in Section 4.6.

The Behavior Mix, which defines the relative frequencies of the extracted Behavior Models, can be obtained in a similar way. By now, we assume that only one user type exists. Even varying workload intensity is still an open issue. The Markov4JMeter add-on for JMeter provides an input option for a formula which describes the possibly varying number of virtual users in a workload generation process. The core challenge is the identification of different sessions in usage records, allowing the extraction of an appropriate formula, analogous to the extraction of Behavior Models.

3.7 Workload Generation

This section describes the generation of synthetic workload for our performance tests, resulting from the workload generation model as described before. The idea of the generation process will be illustrated at first, exposing the derivation of synthetic workload from previously discussed approach artifacts. Afterwards, we specify some input criteria, which must be met by any appropriate workload generation tool for a successful implementation of the introduced approach.

As illustrated in Section 2.2, the synthetic workload to be sent to the SUT shall comply with the workload generated by a population of real users. The described workload model is intended to meet this target by emulating user behavior, which results from an appropriate set of Behavior Models. Each Behavior Model is defined on a certain set of states, associated with use cases. For each use case, its related input flow regarding to the SUT is provided by a table, which has been introduced in Section 3.2. The input flows define the sequences of requests to be sent to the SUT, whenever a related use case shall be emulated. Substitution of Behavior Model states with their related input flows results in a probabilistic rather than sequential input flow. Figure 3.7 illustrates a probabilistic input flow by example, based on the table depicted in Figure 3.2. Probabilities in this figure correspond to the example of a Behavior Model contained in Figure 3.1. The selection of a next use case to be emulated depends on these probabilities. The input flow starts with an execution of use case 0.01 login. With a probability of 40\%, denoted by transition label 0.4, the next use case to be executed is 1.03 delete item. Analogous, an execution of use case 1.01 add item respectively 1.02 edit item might follow with a probability of 30\%. The load generator sends requests, which correspond to the input flow of the currently selected use case. The processing of the model continues, until the exit state $ is reached. Each virtual user processes the model that way, indicating an individual process assigned by a workload generation tool. The Behavior Mix selects the Behavior Models to be emulated with regard to their relative frequencies. For a further approximation to real user behavior, appropriate user think times need to be modeled. Emulation of a given think time means...
3. Performance Testing Approach

that the related load generation process remains in idle mode for a corresponding time, before it sends the next request. This has been already discussed in context of the ON/OFF model in Section 2.3.2.

An appropriate workload generator must provide input options according to the components of the discussed workload generation model. Consequently, it should be able to handle multiple Behavior Models, including the assignment of request sequences to their graph nodes. Furthermore, definitions of guards and actions regarding to an Application Model must be supported, as well as the definition of the Behavior Mix, which specifies the relative frequencies of the Behavior Models. Finally, a load generator must even provide input options for the think times. The Markov4JMeter [van Hoorn et al., 2008] add-on for JMeter meets all of these requirements, since the described model results from it. Hence, we will use this tool for our tests in Chapter 5.

The next chapter illustrates an implementation idea for our approach. It exposes the technical realization using the performance testing tool JMeter and gives an introduction to so-called test plans as well as to the JMeter add-on Markov4JMeter, which is a base part of our implementation. The chapter describes the single implementation steps on a certain abstraction level, so that the implementation with alternative techniques can be derived easily from this approach.
Model-Driven Implementation of the Performance Testing Approach

This chapter illustrates the ideas for a model-driven implementation of the approach which has been introduced in Chapter 3. The procedure does not refer solely to a certain environment nor a defined set of tools, instead the principal steps for a successful implementation will be described. Nevertheless, we declare the performance testing tool JMeter [The Apache Software Foundation, 2013b] as our load generator of choice, since it fulfills with its add-on Markov4JMeter [van Hoorn et al., 2008] the requirements discussed in Section 3.7. The model-driven part of the implementation uses the Eclipse Modelling Framework (EMF) [The Eclipse Foundation, 2013].

Section 4.1 discusses some general issues about the technical realization of performance tests with JMeter, including an introduction of the Markov4JMeter add-on. The remaining sections expose a model-driven implementation process for our approach. The process consists of six single tasks, corresponding to the tasks which have been introduced in Chapter 3. It starts with the manual specification of use cases in Section 4.2 and continues with the manual specification of an application model in Section 4.3. Section 4.4 discusses the generation of an appropriate template via MarkovJMeter for storing Behavior Models. In Section 4.5, concrete instances of user session records received through a dynamic analysis of a production system will be discussed. For the extraction of probabilistic transition values from these records, an appropriate EMF model will be introduced in Section 4.6. Finally, Section 4.7 illustrates the generation of synthetic workload via JMeter, according to the extracted Behavior Models.

4.1 Performance Testing with JMeter and Markov4JMeter

The JMeter tool is a Java based performance testing tool, which supports many different server types, for example, Web-HTTP, HTTPS, or SOAP. The Apache Software Foundation [2013b] states that JMeter’s full multithreading framework allows concurrent sampling by many threads and simultaneous sampling of functions by separate thread groups. The tool provides an API for its functionality being extended by add-ons. Performance testing in JMeter depends on so-called test plans, which consist of elements that provide miscellaneous functionality, e.g., logic control, HTTP requests, or result processing. These elements can
be used to define sequences of user actions, to be emulated by virtual users. A virtual user is represented as an individual thread, which processes the defined test plan for generating corresponding synthetic workload.

The idea of test plans including some of their fundamental elements will be further discussed in Section 4.1.1. The Markov4JMeter add-on for JMeter, which provides an additional set of test plan elements for generating probabilistic workload, will be introduced in Section 4.1.2.

### 4.1.1 JMeter Test Plans

A test plan in JMeter consists of a set of elements, each one associated with a certain action, e.g., sending a request to the SUT, or parsing response data received from the SUT. Those elements are arranged hierarchically, indicating a tree structure. In general, the root element of a test plan is a **Thread Group**, which manages the number of virtual users, respectively corresponding threads. Its tree node contains the elements to be processed by each thread, as child nodes. The number of virtual users is arbitrary and may even vary while workload generation. Figure 4.1 shows the JMeter user interface with an example test plan. The magnified part depicts a Thread Group with a so-called **HTTP Cookie Manager** and a **HTTP Request Defaults element** as well. While the HTTP Cookie Manager defines the handling of cookies, the latter one defines several default values for HTTP requests. Furthermore, the example test plan contains a **Markov Session Controller**, which itself contains a list of
child elements. Besides a timer element named *Think Time Configuration*, the list contains so-called *Markov States*, whose names correspond to the example use cases defined in the table of Figure 3.2. That is, a Markov State can be associated with a use case, corresponding to the states of the Behavior Models discussed in the previous chapters. Consequently, each Markov State mainly consists of *HTTP Requests*, so-called *Samplers*, which can be associated with the actual requests made to the SUT for applying the related use case. The Markov Session Controller and the Markov States both are elements provided by the Markov4JMeter add-on, which will be discussed in more details in the following section. At this point, we just used them for explanatory purposes in terms of a general test plan structure. There is a large number of additional standard elements provided by JMeter.

For the definition of requests to be sent to the SUT, the target parameters must be known. For example, appropriate HTTP-Requests to a Web-based SUT with corresponding input values can only be constructed, if the parameter names for the related form input fields are known. If the SUT has black box character, that is, no information about its internal structure respectively its code is known to the tester, a code analysis is impossible. For such a case, JMeter provides an element with proxy server functionality for filtering out request information. This will be further discussed in the context of a concrete test system in Chapter 5.

### 4.1.2 Markov4JMeter Add-on for JMeter

The Markov4JMeter add-on for JMeter supports additional functionality for using the workload generation model as described in subsection 2.4.3. Therefore, it provides two additional test plan elements, namely “Markov States” and “Markov Session Controller”, for the input of application model artifacts respectively input options for a time-dependent varying number of virtual users.

A *Markov State* represents a state of a Behavior Model. It provides additional input options for guards and actions of outgoing transitions, according to the Session Layer of the related application model. Figure 4.2 depicts the input form of a Markov State, with some example destination states. The *Guard* column accepts boolean expressions in context of certain variables, and the *Action* column accepts related actions. If no guards and actions are explicitly defined for a certain transition, it is assumed that the transition is always valid. The check boxes of the *Disabled* column specify, whether the assigned guards and actions of the related row shall be activated or not.

A *Markov Session Controller* supports the definition of relative frequencies of Behavior Models as well as input for varying workload intensity. The related input form is depicted in Figure 4.3. It shows the assignment of relative frequencies for some example Behavior Models in the *Behavior Mix* sub form. Note that the sum of all relative frequencies must be 1, as discussed in Section 2.4.3. Furthermore, names can be assigned to the Behavior Models for overview reasons. Each Behavior Model is stored as a transition probabilities matrix on a common set of states. The format corresponds to the templates which have been illustrated in Section 3.4. Hence, each matrix is stored in a *comma-separated values*
4. Model-Driven Implementation of the Performance Testing Approach

Figure 4.2. Input form for a Markov State in Markov4JMeter

(CSV) file. The amount of Behavior Models, that is their related CSV files to be included to the controller is unlimited. The Markov Session Controller even supports the generation of a template CSV file for storing Behavior Models. This will be discussed in Section 4.4. Figure 4.3 also shows the Session Arrival Controller sub form. This controller controls the current number of sessions in terms of time-dependent varying workload intensity. Further details will not be illustrated at this point, since we will not use this feature in our performance tests of the evaluation part.

The next section discusses the manual specification of use cases, regarding to our implementation idea.

4.2 Manual Specification of Use Cases

As stated in Section 3.2, the behavior specification of a system generally provides a set of domain-specific use cases. These use cases need to be analyzed regarding to their implementation in the system, since they can be associated with corresponding states of the Session Layer graph contained in an application model. The Session Layer defines the valid input sequences of the SUT. Hence, a table of use cases and their assigned input
4.3 Manual Specification of an Application Model

For the application model, we need to consider the related Session Layer and the protocol layer as well. The implementation is done by using JMeter and its Markov4JMeter add-on. The aim is the construction of a load test plan, which contains possible restrictions defined in the Session Layer and protocol characteristics of the SUT as well.

As described in Section 4.1.2, MarkovJMeter provides test plan elements, named Markov States, which represent states with additional input options for guards and actions of outgoing transitions. Since we associate states of the Session Layer with use cases, a Markov State for each use case needs to be added to the related JMeter test plan, with individual guards and actions, according to the restrictions defined by the sessions layer.
4. Model-Driven Implementation of the Performance Testing Approach

The Markov States even provide input options of JMeter standard states, that is protocol-related input for the requests to be sent to the SUT. As stated before in Section 4.1.1, JMeter provides a proxy server element for capturing request information for Web-based systems. Since the protocol characteristics are SUT-related, no further details will be discussed at this point, but we will use the JMeter proxy server for a concrete system in Chapter 5.

The specification of the application model is fundamental for the generation of a related template to be filled with Behavior Model values. The next section illustrates the generation of templates via Markov4JMeter.

4.4 Template Generation

This section describes the generation of templates for storing Behavior Models. As stated in Section 4.1.2, Markov4JMeter supports a corresponding option for generating appropriate templates in CSV format.

In Section 4.3, the specification of the application model included the definition of Markov States in an according JMeter test plan. In context of the application model specification, a Markov Session Controller element needs to be added to the test plan, too. Besides the definition of the Behavior Mix, it even supports the generation of a template CSV file for the defined Markov States. Figure 4.4 shows the user interface of a Markov Session Controller with the corresponding Generate Template button.

![Figure 4.4. Template generation with Markov4JMeter](image)

30
Each generated template file contains the set of Markov States according to the related test plan. The content of the file is structured similarly as described in Section 3.4. The resulting template assigns initially zero-values to all transition probabilities except to the transitions targeting the exit state, which are initialized with 1 by default. That is, the sum of all probabilities in a row is 1 as required for Markov chains. Figure 4.5 shows an excerpt of a CSV file representing a template by example. The states correspond to those defined in the test plan and discussed in Section 3.4.

![Figure 4.5. CSV template file generated by Markov4JMeter, as a rendered spreadsheet (excerpt)](image)

Section 4.6 describes the implementation steps for filling a given template with actual probabilistic values extracted from user session records. Prior to that, Section 4.5 shortly discusses a concrete example of a user session record file obtained through a dynamic analysis of a production system.

### 4.5 Dynamic Analysis of a Production System

In Section 3.5, a generic format for user session records which can be obtained through a dynamic system analysis has been introduced. This section discusses a concrete example of a user session record, according to this format. The dynamic analysis process has been already discussed in Section 3.5, hence we just expose a concrete result at this point.

Figure 4.6 shows an example of a user session records file, which contains records line by line, using colons and semicolons as token separators. In each line, a leading number denotes a unique session identifier, followed by the sequence of services, which have been executed by a user. A service execution is identified by its assigned name in quotes, followed by its start time and end time, both separated by colons and stored as long values, which have been truncated for the sake of clarity.

![Figure 4.6. Example for a user session records file (excerpt)](image)
4. Model-Driven Implementation of the Performance Testing Approach

The next section introduces a class model which can be used for the extraction of Behavior Models regarding to the given file format.

4.6 Automatic Extraction of Behavior Models

In this section, a model-driven implementation approach for the extraction of Behavior Models from a set of user session records will be introduced. It is assumed that the records have been received through a dynamic analysis of a production system and that they are stored in a common format, as discussed in Section 4.5.

The idea of the extraction process has been already discussed in Section 3.6. The process consists of three steps, starting with the construction of an ABMG, followed by the transformation of the ABMG to a RBMG, and closing with the specification of think times. For the implementation of these steps, a data model is required for storing intermediate results as well as the final result. The EMF allows the automatic generation of source code derived from a specified model, hence we introduce an EMF model, which contains all significant components. This is the model-driven part of the implementation. The EMF model is depicted as a UML diagram in Figure 4.7 and shows the classes and associations between them, which will be discussed next.

The UseCaseRepository class represents a repository of use cases defined through the system’s behavior specification. Each use case indicates a certain service provided by the system and therewith an individual node of a Behavior Model graph.

The UseCase class defines the attributes of a use case. Each use case has an explicit identifier, abbreviated as id, denoting for example a numeric value. Furthermore, each use case has a name, which is the textual representation of the use case within a session record. Both attributes are stored as strings.

A use case can either be observed as an executed service within a user session, or it can be associated with a vertex of a Behavior Model graph. Both cases indicate a use case execution. The abstract parent class AbstractUseCaseExecution models this issue with its two child classes ObservedUseCaseExecution and Vertex. A Vertex represents a node of a Behavior Model graph, and ObservedUseCaseExecution represents a use case execution within a user session. The class ObservedUseCaseExecution has two attributes startTime and endTime which together specify the duration of a service by using long values for points in time.

Analogous to the UseCaseRepository, the SessionRepository contains the user sessions which consist of sequences of executed use cases. The class Session has three attributes, whereas id is an explicit identifier, and startTime and endTime together specify the duration of the related user session. The session time is derived from the observed use case executions. The start time of a user session is defined by the start time of its first observed use case execution, and the end time is defined by the end time of its last observed use case execution. It is assumed that the start times and end times of the observed use cases within a session do not overlap. The identifier is given as a unique number, which identifies a single record as already discussed in Section 4.5.
4.6. Automatic Extraction of Behavior Models

A vertex might have outgoing transitions within a Behavior Model graph. On the other hand, each outgoing transition leads to an adjacent vertex, namely targetVertex. This issue is modeled through the associations between the classes Vertex and Transition. The value of a transition represents the value of its label, which is an absolute or relative double value, indicating the type of the Behavior Model graph.

The class AbstractBehaviorModelGraph is an abstract parent class for the two types of Behavior Model graphs, namely BehaviorModelAbsolute and BehaviorModelRelative. The latter one must follow the constraint that its transition values are between 0 and 1.

The introduced EMF model can be used for the implementation of a related extraction tool. The tool needs to parse the session records regarding their format for analyzing the use case executions in the described way. The EMF model provides the structure for...
4. Model-Driven Implementation of the Performance Testing Approach

storing an ABMG and a derived RBMG as well. Having gained the RBMG, its transition
probabilities can be written to the template generated by Markov4JMeter as described in
Section 4.4. The probabilities of the exit state $ need possibly to be recalculated for an
adjustment to the new probabilities. This has been already described in Section 3.6. In case
the PS and the SUT do not denote the same system, the use case names of both system’s
specifications probably differ. The implementation of an additional name mapping solves
this issue.

The next section discusses some implementation issues for the workload generation
with JMeter.

4.7 Workload Generation

The implementation activities for the workload generation mainly consist of the construc-
tion of a JMeter test plan under the use of the Markov4JMeter add-on, as discussed in
Section 4.1.1. The test plan has to contain the Markov States, which define the requests to
be sent for executing the related services. The elements of a test plan will be processed
by each JMeter thread as described before. JMeter provides many further elements for
additional functionality, e.g., assertions to ensure that correct response data is received. A
detailed discussion of those elements is out of scope for this thesis. However, these elements
can be added as child nodes to Markov States for enhanced functionality, indicating a
hierarchical order. The View Results Tree element in JMeter displays the request results for
those elements in an order which is indicated through their related test plan nodes. For an
advanced visualization of results, JMeter even provides a Graph element, which displays
the response times as charts. The modeling of think times in a JMeter test plan can be done
by using Timer elements. These elements provide input options for idle times of threads
which process the related test plan.

We make use of the discussed elements in our performance tests exposed in Chapter 5,
which discusses the evaluation of our approach. We will introduce a concrete test case and
analyze the test results with regard to their validity.
Chapter 5

Evaluation

This chapter evaluates our approach, based on a case study. The case study environment has been provided by the partners of the DynaMod project. It includes a production system and a system-under-test, according to the systems included in our approach. Both systems will be described in Section 5.1. The subsequent part of the evaluation is divided into a qualitative evaluation and a discussion of a methodology for quantitative evaluation. In the qualitative evaluation, we demonstrate that the implementation ideas for our approach as illustrated in Chapter 4 are practical, targeting session-based systems. In the quantitative evaluation part, we discuss several metrics which have been identified for determining the validity of our approach. Furthermore, we present a methodology for measuring these metrics.

5.1 AIDA Case Study Setting

This section describes the systems which we considered in our case study. As stated before, we distinguish between a production system (PS) and a system-under-test (SUT), according to the overview given in Section 3.1. The PS is an application, which has been already in practical use since several years. It provides use cases from its behavior specification, as well as usage information for our workload generation model. The SUT is a modernized prototype of the PS and will be targeted by our performance tests. In the following sections, both systems will be shortly described.

5.1.1 Description of the Production System

The PS is an information management and retrieval system for inventory data of historical archives. It is written in Visual Basic 6 (VB6) and conforms with a “fat client” architecture style. Its underlying database is based upon different versions of Microsoft SQL Server and the Microsoft SQL Desktop Engine (MSDE). The size of the PS is quite small, as it consists of about 15600 lines of code and eight database tables.

5.1.2 Description of the System-under-Test

The SUT named AIDA-Gear is a prototype for a modernized version of the PS, implementing only a subset of the PS use cases. It is a Web-based application, which is based upon the
5. Evaluation


The Web interface of AIDA-Gear consists of a left-side navigation menu and an adjoining main frame, which displays the currently selected input forms. The selection of an item in the navigation menu opens an associated new form in the main frame, which is put as an active form over the previously opened forms. For each opened form, a related entry is added to a process list below the navigation menu. This list can be accessed by the user for reactivating a previously opened form. A form can be closed by clicking on a corresponding symbol of its upper right corner. In that case, the form will be removed from the main frame, and its related entry will be deleted from the process list.

The user must be signed in to the system for executing any services which might imply a modification of database records. However, a user is allowed to browse through the Web interface without being signed in. Figure 5.1 shows a screenshot with an opened form.

![Figure 5.1. Screenshot of the AIDA-Gear application](image)

The next section illustrates the application of our approach on the given systems.
5.2 Qualitative Evaluation

This section describes how we applied our approach to the given test environment for conducting our performance tests with JMeter and Markov4JMeter. Initially, we analyzed the PS use cases which were provided by the behavior specification of the PS and identified those, which are even implemented in the SUT. Additionally, the input flow for each implemented use case has been determined, serving as basis for the request sequences to be constructed. The Application Model has been modeled in JMeter, represented through a test plan which takes account into technical aspects of the SUT. Therewith, we could generate a template for the Behavior Models to be extracted from a given set of PS usage traces, which were obtained through a dynamic analysis of the PS. The transition probabilities of the Behavior Models have been modeled into the JMeter test plan afterwards. The generation of workload includes estimated think times, which have been added to the JMeter test plan, too. The following sections illustrate details regarding to the single steps.

5.2.1 Manual Specification of Use Cases

The behavior specification of the PS contains 30 use cases, divided into seven packages. These use cases are specified by their name, a trigger information, and a description of their outcome. We identified those use cases, whose functionality is already implemented in the PS and the SUT respectively. Their necessary input flows depending on the input forms of both systems have been determined, including request sequences for any varying use case scenarios. The use cases which are not implemented in both systems have been filtered out. This corresponds to the task described in Section 4.2. The resulting table contains 13 use cases with their related input flows, two of them with two, respectively three varying scenarios.

5.2.2 Manual Specification of an Application Model

The Application Model, consisting of a Session Layer and a Protocol Layer, has been implemented in JMeter by constructing a related test plan. The test plan builds on elements which are provided by JMeter and the Markov4JMeter add-on as well.

The Session Layer is modeled through a test plan, which contains Markov States corresponding to the use cases implemented in the SUT. Each Markov State contains its necessary execution requests as child nodes. The use cases and their related input flows are given through the table, which has been constructed in the previous task. As stated before, the table includes information about use case variants, according to different use case scenarios. For these variants, Markov States have been added to the test plan, too. All Markov States have been configured according to the guards and actions derived from the Session Layer of the Application Model.

The Protocol Layer contains SUT-related information, e.g., path specifications for HTTP requests. These specifications can even be anchored in a JMeter test plan. Since the SUT
5. Evaluation

had “black box” character for us, that is no code information was available, we made use of the proxy server functionality provided by JMeter. The Proxy Server element in JMeter allows the observation of requests made to the SUT and reveals the transferred protocol data. Therewith, we configured our test plan regarding to the technical operations of the SUT.

A special challenge was given through the fact that the SUT is based upon the Java-Servlet specification. Hence, it uses a JSF view state parameter, namely jsfViewState, whose varying value is passed from one request to the next. That is, the parameter value, defined in a hidden field of each HTML body received from a request, needs to be read from this field for its next transfer. JMeter provides a Regular Expression Extractor, which solves this issue. As indicated by its name, this element takes a regular expression and parses the request result data for it. The matching tokens can be named as variables, for being included into any other JMeter request definitions.

For all form input values to be sent to the SUT, we were able to use time stamps in combination with self-defined counter variables, since the SUT does not verify the input values. JMeter provides appropriate elements, predefined time stamp functions and counter variables in particular.

The resulting test plan contains 17 Markov States.

5.2.3 Template Generation

A Markov Session Controller has been added to the JMeter test plan. This element has been already discussed in Section 4.1.2. It supports input options for the relative frequencies of Behavior Models and the option to generate a template for Behavior Models in particular. The template generation via Markov Session Controller has been already described in Section 4.4. The resulting CSV-file represents a probability matrix for the 17 modeled Markov states.

5.2.4 Dynamic Analysis of the PS

A dynamic analysis as described in Section 3.5 has been conducted on the PS. For this analysis, the PS has been instrumented, and usage data has been collected over ten days from two terminals. As a result, we obtained a file of 10 user traces, which serves as basis for the automatic extraction of Behavior Models. Its format complies with the format introduced in Section 4.5.

5.2.5 Automatic Extraction of Behavior Models

For the automatic extraction of Behavior Models, we used the EMF model which has been introduced in Section 4.6, as a basis for the implementation of a related tool. The resulting tool parses the session traces and fills the given template with probability values for each session. Since we do not use any clustering methods and assume that the behavior
characteristics of all users are equal, we obtained one automatically extracted Behavior Model from the session traces of the previous task.

### 5.2.6 Workload Generation

The generation of workload builds on the Markov4JMeter model, which contains the Application Model, the Behavior Models, and a Behavior Mix. The Application Model has been already discussed in Section 5.2.2. Hence, we will only illustrate the steps done for the implementation of the Behavior Models and the Behavior Mix.

As described in the template generation step, a Markov Session Controller has been added to the JMeter test plan for generating a template for Behavior Models. Our Behavior Model extraction tool created an instance of the template with transition probabilities, according to the values of the Behavior Model which has been extracted from the session records. Consequently, the automatically extracted Behavior Model has been stored in an own CSV file. Besides the generation of a template, the Markov Session Controller provides input options for the relative frequencies of Behavior Models. The relative frequency of the extracted Behavior Model has been set to 1.0 in the Markov Session Controller.

As mentioned before, we defined the think times manually. Therefore, an average think time delay has been estimated, including a deviation value. We chose for each Markov State a constant delay of 2000ms with a deviation of 500ms. This has been modeled by using a Gaussian Random Timer element in JMeter, which generates according random values.

The performed steps led to a test plan which can be processed by JMeter for generating workload, which corresponds to workload of real users. The virtual user’s behavior results from the usage data of the PS. The number of virtual users is arbitrary, but for simplification, we set the number of virtual user to 1 in the remaining evaluation part. After demonstrating that the approach can be applied to session-based systems, we discuss its quantitative evaluation in the next Section.

### 5.3 Methodology for Quantitative Evaluation

In this section, we discuss the quantitative measurement of the validity of our approach. Thereby, we define validity as the amount of deviation between the workload generated by real users to be emulated and the purpose adapted synthetic workload. For the determination of the validity, we identified several evaluation metrics to be considered for our approach. This will be discussed in Section 5.3.1. In Section 5.3.2, we present a methodology for evaluating our approach in terms of these metrics.

#### 5.3.1 Evaluation Metrics

This section discusses several metrics which have been identified for the evaluation of our approach. First, we illustrate the characteristics of “appropriate” synthetic workload. The
5. Evaluation

metrics which will be discussed afterwards indicate the level of achievement. To determine whether the synthetic workload meets the goal of generating workload of real users, the focus needs to be put on the following points:

 Pública

Representativeness of user sessions

The sequences of service invocations realized by virtual users must hold the same characteristics as indicated by the session traces of real users from which the sequences have been derived. Therefore, the Behavior Models extracted from the traces must characterize corresponding navigational patterns.

Publica

Representativeness of (possibly varying) workload intensity

The temporal distribution of concurrent virtual users in a workload generation experiment should correspond to the temporal distribution of real users using the considered system in the same time range. For example, if a certain number of real users access the system in a certain time range sequentially, the virtual users should preferably access the system in the same time range similarly.

For the representativeness of user sessions, the following metrics need to be considered, since they indicate whether the behavioral characteristics of virtual users comply with those of real users:

 Pública

M1. Number of requests

The number of requests generated by virtual users must correspond to the number of requests generated by real users. This includes all requests required for transferring any data objects, e.g., documents or images.

 Pública

M2. Session length

The length of a session is defined as the number of its (invoked) services. The average length of sessions instantiated by virtual users of a certain user type must correspond to the average length of sessions instantiated by real users of the same user type.

 Pública

M3. Think times

The time periods between the invoked services of virtual users must correspond to the think times of real users.

For the representativeness of (possibly varying) workload intensity, server logs resulting from synthetic workload generation need to be analyzed in terms of overlapping sessions for detecting the number of virtual users at certain points in time. Therewith, a corresponding formula which describes the possibly varying number of virtual users, can be figured out to be compared with the formula used for synthetic workload generation. In general, varying workload intensity is still an open issue in our approach, as already discussed in Section 3.6. However, the compliance of synthetic workload with workload of real users in terms of M1, M2, and M3 should imply a system usage which corresponds to the system usage of real users to a certain degree, referring to typical metrics like, e.g., CPU utilization, Memory utilization, and Disk usage.
5.3. Methodology for Quantitative Evaluation

5.3.2 Approach for the Comparison of Metrics

For the comparison of the metrics discussed in Section 5.3.1, the session identities and request time information must be provided by the server logs. In general, Web servers like, e.g., Apache Tomcat, can be configured in such a way that the server logs include this information as session-ID and request timestamp, respectively. For the extraction of service information as required for $M_2$ and $M_3$, a procedure analogous to the dynamic analysis of a production system is necessary. Instead of analyzing PS user session traces, server logs of the SUT need to be analyzed, aiming to the session-related extraction of invoked services from request sequences. This is an open issue for future work, but we introduce the idea at this point. Hereinafter, we assume that the Behavior Models of a workload generation model are known through predefinition or extraction from user traces. For determining the validity of the workload generation model, the following three preparation steps need to be done:

1. **Calculation of average step numbers in Behavior Models**
   Representing the expected session length for a specific user type, the average number of steps from the initial state to the exit state of the related Behavior Model needs to be calculated.

   Each Behavior Model indicates a Markov chain which is defined on a set of states, including transitions of probabilities, initial state, and exit state. Transition matrices which represent a Markov chain indicated through these components, have been already discussed in Section 3.4. The average number of steps from the initial state to the end state of a Markov chain can be calculated algorithmically by setting up and solving a linear equation system, based on the related transition matrix. Related work can be found in Gallager [1995, p.119 ff.].

2. **Construction of a transition model for available services**
   For the identification of invoked services in a sequence of requests, the request structure of each service must be known. This can be achieved by a controlled execution of use cases in the SUT, including an analysis of the resulting server logs. The step is similar to the controlled execution of use cases within the dynamic analysis of a production system, as described in Section 3.5. As a result, a transition model can be constructed for the identification of services in sequences of requests.

3. **Construction of acceptors for Behavior Models**
   As stated before, we assume that the Behavior Models of an experiment are previously known. For the identification of user types from session sequences, appropriate acceptors for the Behavior Models must be constructed. These acceptors must be derived under consideration of the transition probabilities, since these values might indicate different Behavior Models.

Suppose we start an experiment with Markov4JMeter, targeting the SUT with a workload generation model. The resulting server log can be parsed for requests which belong to
5. Evaluation

an individual session. With the constructed transition model for available services, we can identify the invoked services in a sequence of session related requests. Therewith, we obtain comparable session lengths by performing the following five steps:

1. Identification of session related requests
   The requests contained in a server log can be assigned to their related sessions via session-ID. Consequently, each session-ID indicates an own sequence of requests.

2. Identification of session related services
   For each sequence of session related requests, the indicated invoked service can be identified under use of the constructed transition model for available services. Hence, we obtain the sequence of invoked services for each session.

3. Identification of user type
   For each sequence of sessions, the indicated user type can be identified under use of the previously constructed acceptors for Behavior Models. As a result, we obtain the Behavior Model of the (virtual) user type who instantiated the related session.

4. Calculation of average session lengths
   The number of session related services simply corresponds to the session length. The average length of a user type specific session can be calculated from the lengths of all sessions instantiated by virtual users of same type.

5. Extraction of user think times
   From the service related sequences of requests, the user think times can be extracted, too. For a service invoked in a session, its user think time corresponds to the time interval between the first service related request and the last request of the previously invoked service.

The average session length and average session length of each user type can be compared to each other. The extracted think times can be compared to the expected think times, too. Our thesis is that the values become closer to each other, the longer an experiment with a given workload generation model runs. For proving this, the remaining open points need to be completed at first. This is a future work issue and will be further discussed in Section 6.3.
Conclusions and Outlook

6.1 Summary

In this paper, we introduced a model-driven performance testing approach including its evaluation. The approach builds on Markov4JMeter’s analytical workload generation model, which aims to the generation of probabilistic workload. The introduced approach consists of six tasks, starting with an analysis of use cases for building an Application Model based upon them. The Application model provides a template for probabilistic transition values of Behavior Models. These values, and therewith the Behavior Models themselves, can be extracted from user traces, retrieved through a dynamic system analysis. The extraction of Behavior Models represents the model-driven part of our approach. The resulting workload model is the basis for the workload generation process itself, which is the final task. Furthermore, we illustrated our implementation ideas for the approach.

In the qualitative part of the evaluation, we demonstrated that the implementation ideas for our approach are applicable, targeting session-based systems. In the remainder of the evaluation, we discussed several metrics for a quantitative evaluation of the approach and identified several open issues for future work. Furthermore, we presented a methodology for measuring these metrics, aiming to the proof that any results gained through the application of our approach are reasonable.

In Section 6.2, several challenges which we discovered in the evaluation part will be discussed. Section 6.3 summarizes the future work issues.

6.2 Discussion

Our approach includes several challenges, which will be discussed in this section. For some of these challenges, solutions have been already given in the description of the tasks, others are still open for future work.

The specification of the use case table in the first task of our approach requires high effort for covering possible scenarios of each use case. In our case study, the amount of use cases is relatively small, since the considered test system is small. Consequently, the amount of possible scenarios remains small. For a large system, the manual specification of all scenarios is hardly possible, since their number might increase rapidly. In that case, a
model-driven generation of use case scenario descriptions is desirable. The generation via system screen flows constitutes a future work issue.

The manual specification of an Application Model includes the definition of a Protocol Layer, which requires protocol information of the considered system. If this information is not available, that is, the system is a “black box” for the tester, it can only be obtained through a dynamic analysis. This generally includes a trial-and-error approach, since certain properties, e.g., parameter names or valid parameter values, are initially unknown. In our case study, we discovered the JMeter HTTP Proxy Server as being of great help. The user interface of JMeter additionally provides drag-and-drop functionality, which simplifies the configuration of this element.

Our approach presumes that usage data of the considered test system is available. In the case study, user traces have been retrieved from an instrumented production system, which is the legacy variant of the system-under-test. The user traces supplied the transition values of the Behavior Models, that is probabilities of Markov chains. It is important to note that the estimation of those probabilities is difficult otherwise. Hence, these values should be extracted from usage data, if available. For a system with no usage information, e.g., a new application which has not been in productive use before, a similar system could be instrumented for retrieving appropriate user traces.

For the workload generation, a related JMeter test plan has been constructed. The test plan addresses protocol-specific SUT properties, e.g., parameter names. Consequently, it is fragile with respect to protocol changes. To solve this issue, the generation of sampler templates for MDSD-based applications is desirable. This point is even declared as future work.

6.3 Future Work

This section points out the main future work issues, which concern the improvement of the workload generation model as well as the simplification of the modeling process. Furthermore, the quantitative evaluation of our approach is still one of the core challenges.

- The automatic extraction of think times from a given set of user traces is still an open implementation issue. If time information is available as in our sets of user records, the extraction can be done straightforward. In the implementation of our approach, the user think times still have been estimated.

- The determination of the Behavior Mix from a given set of session traces is still an open implementation issue, too. In our approach, we assumed that only one user type exists. If several user types can be identified from session traces, their relative frequencies of occurrence needs to be determined, too.

- The (possibly varying) workload intensity cannot be extracted from session records in our approach yet. Therefore, the number of sessions at several points in time
6.3. Future Work

within an experiment needs to be identified in the session traces. These measurement points indicate a function from which a polynomial function can be obtained through approximation. The input option for a polynom in Markov4JMeter already exists. However, the measurement and approximation part still needs to be done.

- Storing the user think times is not provided by the Markov4JMeter add-on yet. In our case study, the think times have been modeled as random values of certain range in the JMeter test plan. As a solution for including them directly into the Markov4JMeter model, they could be stored as additional entries in the cells of the Behavior Model matrices. Therewith, the entries of the resulting matrices need to denote tuples of two elements instead of single probability values.

- As discussed before, the model-driven extraction of use case scenarios is still an open issue. The SUT defines with its navigational patterns certain sequences, which might be indicators for behavioral issues, for example, login forms as conditions for further input. One challenge is the extraction of this information, e.g., from screen flows of the system.

- For generating a test plan which is not fragile with respect to protocol changes, though including protocol-specific SUT properties, the generation of sampler templates for MDSD-based applications is desirable.

- We assumed in the description of our approach that a session trace indicates exactly one user type. In general, session records result from multiplate users, and consequently, they indicate different user types. For the identification of user types and their related Behavior Models in particular, clustering methods as introduced by Menascé et al. [1999] can be used as an appropriate method. This still constitutes an open issue for our approach.

- The answer for the quantitative validity of our approach is still a core challenge. Therefore, a methodology for measuring certain metrics has been introduced, consisting of several steps. The implementation issues for these steps still need to be solved, that is calculation of expected steps in Markov chains and construction of transition models in particular.

- The validity of JMeter test plans which result from our approach even needs to be determined. Therefore, our future research work aims to a measurement-based evaluation with an existing generic application.


Bibliography


