Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Declarative User Experience Regression Analysis in Continuous Performance Engineering

Manuel Palenga

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Dr.-Ing. André van Hoorn Dr. Dušan Okanović Henning Schulz, M.Sc. Vincenzo Ferme, M.Sc.
Commenced:	Juli 21, 2018
Completed:	December 21, 2018
CR-Classification:	D.4.8, G.3

Abstract

Software performance has a major impact on the user satisfaction. When users have to wait longer as they did in a previous software versions, profit could be creased as users could leave the system. Ensuring user satisfaction deals with regression analysis on the system level. A regression analysis can compare the performance of several system versions. For the comparison, measurements from the system level are needed. The measurements can be collected by load tests with representative user behaviors in order to represent the real system load as accurate as possible. For the definition of a regression test that consists of a regression analysis, a language is needed that supports the definition of the user concerns and not the execution flow itself. This language simplifies the creation of regression tests. The declarative DSL is such a language that can be used for this purpose. However, no approach exists which allows a declarative language to define a regression test consisting of a regression analysis for representative load test measurements.

The aim of this thesis is to provide such an approach. The thesis envisions an approach consisting of the tools BenchFlow and ContinuITy, which support subsets of the required functionalities. The declarative DSL from BenchFlow is used to define the regression tests including the user concerns, and ContinuITy provides the corresponding representative user behaviors. The thesis introduces a workflow for the automated execution of the regression tests. This includes the deployment of the systems to be tested, the execution of the load tests, and the regression analysis to compare several system versions. An evaluation of the regression analysis with a representative application shows that regressions can be detected correctly. A further evaluation of the BenchFlow DSL shows that the BenchFlow DSL extensions, which are outcomes of this thesis, have advantages and disadvantages compared to related models. With this approach, performance experts and software engineers can write and execute regression tests with low effort to ensure software performance.

Kurzfassung

Die Software-Performance hat einen großen Einfluss auf die Zufriedenheit von Nutzern. Sobald Nutzer länger warten müssen als bei einer vorherigen Software Version, kann das dazu führen, dass die Nutzer das System verlassen, wodurch weniger Profit gemacht wird. Die Zufriedenheit der Nutzer kann durch eine Regressionsanalyse auf der Systemebene sichergestellt werden. Die Regressionsanalyse kann die Performance von verschiedenen Systemversionen miteinander vergleichen. Für den Vergleich werden Messwerte von der Systemebene benötigt. Die Messwerte können durch Lasttests erhoben werden, wobei die Lasttests ein repräsentatives Nutzerverhalten enthalten. Dieses Nutzerverhalten wird benötigt um eine möglichst präzise Systemlast zu generieren. Eine Regressionsanalyse kann Bestandteil von einem Regressionstest sein. Für die Definition eines Regressionstests wird eine Sprache benötigt, welche es ermöglicht die Benutzerinteressen anstatt des Kontrollflusses des Regressionstests zu definieren. Mit so einer Sprache wird die Erstellung von Regressionstests vereinfacht. Für diesen Zweck kann eine deklarative DSL verwendet werden. Zum aktuellen Zeitpunkt ist kein Ansatz bekannt, welcher es ermöglicht mit einer deklarativen Sprache einen Regressionstest zu definieren, wobei der Regressionstest aus einer Regressionsanalyse für die Messwerte von Lasttests besteht.

Das Ziel dieser Arbeit ist es, einen solchen Ansatz zu entwickeln. Diese Arbeit stellt einen Ansatz vor, welcher aus den Anwendungen BenchFlow und ContinuITy besteht. Diese Anwendungen werden benötigt um Teilfunktionalitäten für diesen Ansatz bereitzustellen. Die deklarative DSL von BenchFlow wird verwendet um den Regressionstest und die Benutzerinteressen zu definieren. ContinuITy wird verwendet um ein repräsentatives Nutzerverhalten zu erhalten. Des Weiteren wird in dieser Arbeit ein Ablauf zur automatisierten Ausführung eines Regressionstests erklärt. Dieser Ablauf enthält die Bereitstellung des zu testenden Systems, die Ausführung der Lasttests und die anschließende Regressionsanalyse um verschiedene Systemversionen miteinander zu vergleichen. Eine Evaluation mit einer repräsentativen Anwendung ergab, dass die Regressionsanalyse unterschiedliche Regressionen korrekt erkennen kann. Für die zweite Evaluation wurden die BenchFlow DSL Erweiterungen, welche im Rahmen dieser Arbeit gemacht wurden, mit verwandten Modellen verglichen. Dabei konnten die Vorteile und Nachteile der Erweiterungen abgeleitet werden. Mit dem entwickelten Ansatz ist es möglich, dass Performance Experten und Softwareentwickler mit wenig Aufwand einen Regressionstest schreiben und ausführen können, sodass die Software-Performance sichergestellt werden kann.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

BPMN Business Process Model and Notation

CI Continuous Integration

CSV Comma-separated Values

DSL Domain-specific Language

GUI Graphical User Interface

JSON JavaScript Object Notation

LOC Lines of Code

REST Representational State Transfer

SUT System Under Test

XML Extensible Markup Language

List of Listings

Chapter 1

Introduction

1.1. Motivation

Continuous delivery [HF10] is an increasingly important method for releasing new software application versions frequently to customers and users. Each version has to be tested before the release to ensure that the application works as required. Mainly unit, integration, and system tests are used to ensure the software quality [JH15]. With the mentioned types of tests, it is possible to test functional and non-functional requirements.

The performance is an important non-functional requirement, e.g., Amazon loses 1 % of sales for a 100 ms delay [Lin06]. Testing the software performance with many possible user actions requires load tests which map the real workload. ContinuITy [SAH18] is a tool that provides representative workloads for applications which can be used for load testing. This type of tests is well suited for checking the performance of a service. 81 % of the applications in the ProgrammableWeb API directory use REST as API architectural style [San17].

The result of load tests can be performance metrics, e.g., the response time of an operation. To prove the performance, performance expectations for the application are required. A possible performance expectation is an upper limit of the response time of an operation. When setting an upper limit in a test, the response time has to be lower than the upper limit but a degradation of the response time can remain hidden. For this case, a regression analysis is required which compares the performance of different versions of a software application. An improvement of the response time leads to higher user satisfaction and to more users because they do not leave the web page, e.g., 40 % of the users abandon a web page after a load time of three seconds [Wor11]. The degradation of the performance has negative impacts which can be the case of fewer users and less profit.

This regression analysis requires time because each application version has to be load tested. Also, the definition of the regression test requires time, to simplify the test definition, a declarative language is necessary [HB10]. Existing tools provide a declarative language to define load tests but not in combination with the required regression analysis. For example, BenchFlow [FP17] is a tool which supports the automation of performance tests, but not for regression tests. Also, BenchFlow provides a declarative language for the definition of load tests. Therefore, an automated approach is necessary which provides a declarative language for the definition of regression tests in combination with the regression analysis based on load testing with representative workloads.

1.2. Overview of the Approach

The approach of this thesis is divided into three parts. The first part is the extension of the declarative BenchFlow DSL to support the definition of HTTP workloads. The HTTP workloads are necessary for the execution of load tests. The second part is the workflow of ContinuITy to transform the representative user behavior into the workload model from BenchFlow. This workload model is a part of the BenchFlow DSL. The DSL is necessary for the third part of the approach which is the workflow of BenchFlow. This part consists of (1) the transformation of the HTTP workload into an executable load test from a suitable load testing tool, (2) the deployment of the SUT version, (3) the execution of the load tests for this version, (4) the regression analysis that compares the performance measurements of each version to detect regressions, and (5) the generation of a regression report.

1.3. Goals

The aim of this thesis is to provide an approach for automated regression testing based on load testing with representative user behaviors using a declarative approach. This is divided into the following goals:

Extension of the BenchFlow DSL: The first goal is to extend the BenchFlow declarative Domain-specific Language (DSL) by HTTP workload and regression-specific properties. Load testing of APIs requires the definition of HTTP requests which are part of the HTTP workload. Necessary for the regression analysis are specific properties consisting of the regression goals and the regression conditions. Furthermore, the BenchFlow DSL has to provide the possibility to define deployment properties which are necessary for deploying the System Under Test (SUT) automatically.

Transformation of the ContinuITy model into the BenchFlow DSL The next goal is that ContinuITy has to be able to use the BenchFlow DSL. The HTTP workload and the representative user behavior based on monitored data from ContinuITy has to be transformed into a BenchFlow workload, which is appended to a corresponding BenchFlow DSL.

Regression analysis in BenchFlow: A goal is that BenchFlow has to be able to compare a system with different versions which are specified in the DSL using the defined regression conditions. The analysis has to detect a degradation of the system performance when a degradation exists.

Evaluation of the regression analysis: A further goal is to evaluate the regression analysis using different scenarios. This evaluation has to find false negative and false positive results using SUT versions with and without injected regressions.

Evaluation of the DSL extension: The last goal is to evaluate how good the BenchFlow DSL extensions are compared to related models. The evaluation has to provide findings which contain possible improvements for the BenchFlow DSL.

1.4. Thesis Structure

The thesis is structured in the following chapters:

Chapter 2 - Foundations explains concepts and terminologies which are relevant for this thesis. This includes performance testing and regression testing as well as the tooling of the thesis.

Chapter 3 - Related Work presents research works which are related to the topic of this thesis.

Chapter 4 - Approach describes the BenchFlow extensions including HTTP workload and regression extensions, the ContinuITy workflow including the transformation from the ContinuITy model into the BenchFlow DSL, and the BenchFlow workflow including the regression analysis and the workflow results.

Chapter 5 - Implementation explains the implementation details of Chapter 4.

Chapter 6 - Evaluation shows two different evaluations for the BenchFlow DSL extensions and the execution of the thesis approach. This includes the research questions, the setup, the results, and the result discussion for both evaluations.

Chapter 7 - Conclusion summarizes the thesis and gives an overview of possible future works.

Chapter 2

Foundations

This chapter describes terminologies and tools of this thesis.

First of all, Section 2.1 describes the terminology of *performance testing* and Section 2.2 covers the terminology of *regression testing*. The terminology of *declarative domain-specific language* is explained in Section 2.3. The tools and approaches which are used in this thesis are described in Section 2.4.

2.1. Performance Testing

Performance testing has the goal to determine performance measurements, e.g., response time and resource utilization, from systems in a controlled environment under a predefined load [MFB+07; Sub06].

Meier et al. [MFB+07] mention four common types of performance testing. These are described in the following:

Performance test: The focus of this test type is to determine the performance of the SUT with a single user, e.g., for a single request or for a sequence of requests. With this test, it is possible to adjust the performance expectations [Sub06].

Load test: Load testing checks how a system behaves under normal and peak load. For load tests, representative user behaviors, including the think times between requests, are important to have a representative load. The load can be changed during the load tests, e.g., to represent the course of a normal day [Sub06].

Endurance testing: A subset of load testing is endurance testing [MFB+07]. It checks how well a system is executed in a longer period of time, e.g., multiple days. This test

focuses on resource utilization, e.g., to find memory leaks or uncommitted database transactions [Sub06].

Stress test: Stress testing checks how a system behaves under more load than the normal or peak load [MFB+07]. The system is tested with more users than the system can handle. This test can be done with different load conditions. The test starts from a base load, which is the supposed limit, then the workload is increased in surges in order to simulate more users. These surges lead to peaks with the result that the supposed load limit is exceeded [Sub06].

Capacity test: Capacity testing checks how many users are supported by the SUT under normal and peak load, whereby the performance expectations are fulfilled. Therefore, it is possible to determine if the system has to scale up to support the future load [MFB+07].

Other types of performance testing are the unit test which tests code segments and the component test which tests a single component [MFB+07].

2.2. Regression Testing

The ISO 90003 [ISO 90003] defines regression testing as “testing required to determine that a change to a system component has not adversely affected functionality, reliability, or performance, and has not introduced additional defects”. The tested system is called the System Under Test (SUT).

The behavior includes functional and non-functional aspects, e.g., the performance of a system. For the functional aspects, unit tests are required which test an application or an application component with a predefined expected value. These unit tests are used for each software version to detect behavior changes, called regression.

Software changes can also cause non-functional changes, e.g., performance changes, which can be analyzed with regression testing, e.g., if a request takes more time than the version before, a regression is detected. If the duration of a request decreases, the performance changes with a positive impact and therefore, no regression is detected.

2.3. Declarative Domain-specific Language

The term declarative domain-specific language consists of two different terms, namely the declarative paradigm and the domain-specific language (DSL). Together, the terms

describe a declarative language which can be used in a specific problem domain. In the following, both terms are described.

Domain-specific language: A domain-specific language (DSL) focuses on a specific problem domain. The language can be either a programming language or a specification language. Each language consists of notations which match to the specific problem domain [VKV00].

Declarative paradigm: The declarative paradigm describes "what" the results of an algorithm should be and not how to execute the algorithm. This paradigm splits the user concerns including the goals and constraints from the solution approach [WHK+16]. That enables to separate the specification from the execution with the benefits that the programming task can be simplified and users do not have to learn a complex language. An example language for the declarative paradigm is SQL. In contrast to that, the imperative paradigm describes the "how", e.g., how the control flow looks like. An example language is Java [HB10; PV06].

2.4. Tooling

The approach of this thesis requires tools and approaches to fulfill the goals from Section 1.3. The used tools and approaches are described in this section. Section 2.4.1 describes the WESSBAS approach. In Section 2.4.2, the tool BenchFlow is described, and ContinuITy is described in Section 2.4.3. The load test tool Apache JMeter is described in Section 2.4.4.

2.4.1. WESSBAS

WESSBAS [VHS+18] is an approach to extracting workloads for load testing and performance prediction. It aims to "automate the extraction and transformation of workload specifications for load testing and model-based performance prediction of session-based application systems" [VHS+18]. WESSBAS has three main components.

- (1) A WESSBAS DSL for the specification of the workload. An example is shown in Listing A.1. WESSBAS DSL and WESSBAS model means the same.
- (2) The automated extraction of workloads from recorded session logs.
- (3) The transformation of the workload into executable workload specifications of load testing tools, e.g., Apache JMeter.

The main components of the WESSBAS model are the following:

- **Application model:** It describes the SUT with its corresponding services as application state.
- **Application states:** An application state describes a service. A service consists of (1) properties which are required for a request, e.g., domain, port, and HTTP method, and (2) parameters, e.g., URL parameters or query parameters. (3) Possible transitions to other services are also part of the service.
- **Behavior mix:** The behavior mix represents the proportion of different behavior models. It contains a list of behavior model references with a corresponding probability of the behavior mix proportion. The sum of all probabilities is 1.0, i.e., 100 %.
- **Behavior models:** A behavior model represents a probabilistic user session with invoked services as Markov states. All Markov states in a behavior model form a Markov chain which is explained in Section 2.4.1.1.
- **Markov states:** A Markov state describes a service but without properties and parameters. A Markov states references to the corresponding application state. It contains a list of possible transitions to other services, i.e., Markov states, with their corresponding transition probabilities and think times. The sum of all transition probabilities is 1.0, i.e., 100 %.

2.4.1.1. Markov Chain

A Markov chain is a stochastic model, containing a set of states which are connected with probabilities. These probabilities define the probability that after one state, another state follows. For each state, the sum of all outgoing probabilities is 100 %. With this model, it is possible to define the probability of a transition from one state into another state [MT12].

An example of a Markov chain is shown in Figure 2.1. In this example, the probability that state B follows state A is 50 %, and state D is not dependent on state C and vice-versa. The example contains a last state which is state E because state E has no outgoing transition to another state.

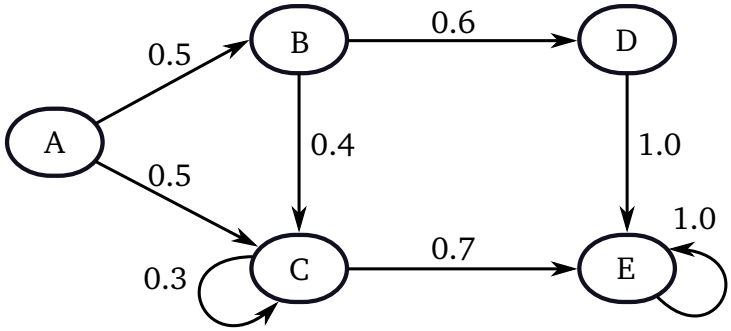


Figure 2.1.: Example of a Markov chain with five states. Start state is A, and end state is E.

2.4.2. BenchFlow

BenchFlow is "a platform for end-to-end automation of performance testing and analysis" [FP17]. BenchFlow is an open-source software, and it is available on GitHub[1]. The core goal of BenchFlow is to define load tests in a declarative manner using a declarative domain-specific language (DSL). It enables to define the complete load test in a single YAML file consisting of the workload, the goals, the system under test (SUT), and the termination criteria [FP18]. An example of a part of the BenchFlow DSL is shown in Listing 2.1, it contains a load test with 100 users, a ramp-up time of one minute which means that the 100 users are created evenly during the minute, and a maximal duration of 30 minutes. The user does not need to know anything about the implementation, because nearly everything the user wants to do can be defined in the BenchFlow DSL. If something is not available in the definition of the DSL, the DSL can be extended [FP18].

A main component of BenchFlow is the test life cycle which contains a nested experiment life cycle. This life cycle manages the test execution. Both cycles are not used in this thesis. Therefore, they are not described, more details are available in the work of Findahl [Fin17].

BenchFlow uses mainly the following four tools:

- **Faban:** Faban[2] is a tool for the definition and execution of performance workloads. BenchFlow uses Faban for the execution of performance tests.

[1]BenchFlow: https://github.com/benchflow
[2]Faban: http://faban.org/

Listing 2.1 An example of a BenchFlow DSL part.

```
configuration:
  users: 100
  goal:
    type: LOAD
  workload_execution:
    ramp_up: 1m
    steady_state: 10m
    ramp_down: 10m
  termination_criteria:
    test:
      max_time: 30m
```

- **Minio**: Minio[3] is a distributed storage server. BenchFlow uses Minio for the data collection including the Faban results.
- **Apache Cassandra:** Cassandra [4] is a *Not only SQL* (NoSQL) database which supports scalability and distribution of data. BenchFlow uses Cassandra as a part of the data analysis.
- **Apache Spark:** Spark[5] is an analytics engine to process large set of data. BenchFlow uses Spark as a part of the data analysis.

BenchFlow supports to execute workloads with Faban, but Faban does not provide all necessary functionalities which are required for this thesis. Faban does not support the extraction of regular expressions. A possible implementation of the extraction would lead to problems with the think times because the think time starts directly after the response. If the extraction requires more time than the think time, that will lead to an exception in Faban with the effect that the complete execution of the workload stops.

2.4.3. ContinuITy

ContinuITy [SAH18] is a tool for "Automated Performance Testing in Continuous Software Engineering" [NTRSS18]. It is a research project with a collaboration of the department Reliable Software Systems Groups of the University of Stuttgart and the NovaTec Consulting GmbH. ContinuITy is an open-source software, and it is available on GitHub[6]. It is a part of the ContinuITy project[7].

[3]Minio: https://www.minio.io/
[4]Apache Cassandra: http://cassandra.apache.org/
[5]Apache Spark: http://spark.apache.org/
[6]ContinuITy: https://github.com/ContinuITy-Project/ContinuITy
[7]ContinuITy-Project: https://github.com/ContinuITy-Project

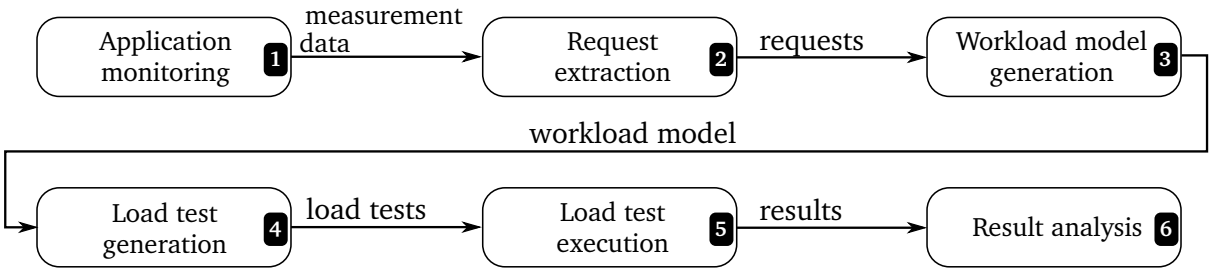


Figure 2.2.: Sketch of the workflow of ContinuITy.

ContinuITy has two goals. The first goal is to test the performance of application services, e.g., REST API, using load tests based on extracted real user behaviors. The second goal is to reduce the number of manual adjustments to execute load tests in comparison to existing approaches. To achieve this, a workflow with different steps is required which is shown in Figure 2.2. For the real user behavior, information is necessary about which interface methods are mostly used, in which order, and what the required input parameters and values are. This information is obtained using application performance management (APM) tools [HHMO17], e.g. Kieker [VWH12] or inspectIT[8], which monitor the whole application and extract measurement data, e.g., method durations and requests (step 1). The HTTP requests are important for load testing. Therefore, the requests have to be extracted with the corresponding order and parameters (step 2). With all requests and the corresponding order, a workload model can be generated. ContinuITy uses the WESSBAS [VHS+18] workload model which provides the advantages of WESSBAS including the creation, the transformation, and the usage of the workload model which is made for load testing and performance prediction. The workload model consists of properties of each request and a Markov chain (see Section 2.4.1.1) with the requests as nodes which are connected if the requests are called in a row and a probability for each transition (step 3). The Markov chain represents the behavior of users. This Markov chain can be used to generated load tests. ContinuITy uses Apache JMeter as the load test tool because WESSBAS provides the functionality to transform its format to the JMeter test plan. The load test contains an order of requests which should be executed. This order describes a representative user behavior (step 4). The load test is executed with Apache JMeter (step 5), and the execution provides results which can be analyzed (step 6).

ContinuITy stores information about a workload in three different models. The first model is the WESSBAS model which is described in Section 2.4.1. The second model is the Input Data Properties Annotation (IDPA) application model. It contains request information including the domain, path, the headers, and which parameter is used with

[8]inspectIT: http://www.inspectit.rocks/

the corresponding parameter type, e.g., URL parameter, but the value of the parameter is not defined in this model. The value is defined in the third model: the IDPA annotation model. It manages all parameter values including single values, list of values, counters, and extractions. An extraction belongs to a specific request, and it consists of a pattern which is used for the response of the request. The extracted value can be used for a later request.

The ContinuITy project consists of the following subprojects:

- **API:** It contains information of the services of each subproject, e.g., the path of a REST method, and the common data formats which are exchanged by the services.
- **BenchFlow:** This complete subproject is a part of this research. The implementation is explained in Section 4.3.
- **CLI:** CLI means command line interface which is provided by ContinuITy. CLI can control all operations which can be executed with the REST API, e.g., the creation of a load test for JMeter.
- **Commons:** This subproject contains auxiliary methods for all other subprojects.
- **Eureka:** Eureka is the service registry.
- **IDPA:** It contains the definition of the IDPA application and the IDPA annotation model. It provides auxiliary methods for the usage of the IDPA models.
- **IDPA annotation:** This subproject consists of the management of the IDPA annotation model. The management consists of the storage and the provision of the models.
- **IDPA application:** This subproject consists of the management of the IDPA application model. The management consists of the storage and the provision of the models.
- **JMeter:** It contains the creation and execution of JMeter test plans which can be triggered by other services.
- **Orchestrator:** This is the central subproject. All orders which should be conducted will be triggered by this project, e.g., the creation of a load test for JMeter.
- **Session logs:** It contains the management of session logs which are extracted by monitoring an application.
- **Wessbas:** This subproject consists of all components which require the WESSBAS model. The WESSBAS model is only known in this subproject and not in the other subprojects. If another subproject needs information from the WESSBAS model, the required information is extracted in this project and are exchanged with a

format defined in the API subproject. This subproject was extended during this research. The extensions are described in Sections 4.3.1 and 4.3.2.

2.4.4. Apache JMeter

Apache JMeter[9] is an open-source application for load testing applications to test the functionality and the performance. JMeter supports different protocols including HTTP. It provides the possibility to test services, e.g., services based on Representational State Transfer (REST) or Simple Object Access Protocol (SOAP), and databases using Java Database Connectivity (JDBC). JMeter is a Java application which provides high portability. It contains a Graphical User Interface (GUI) for the definition of the test plan which generates an Extensible Markup Language (XML) file with the ending *.jmx*. The load test with the test plan can be triggered by the command line. After the execution, JMeter provides a Hypertext Markup Language (HTML) report and the results in a Comma-separated Values (CSV) or an XML file.

JMeter supports extensions. A JMeter extension is *Markov4JMeter*[10] which provides the possibility to define Markov chains including think times for each transition. This extension is used for this thesis.

[9] Apache JMeter: https://jmeter.apache.org/
[10] Markov4JMeter: https://github.com/Wessbas/wessbas.markov4jmeter

Chapter 3

Related Work

This chapter shows the related work of this research. The research approach comprises (1) the selection and determination of workloads, (2) the workload transformation into a predefined declarative DSL, (3) the automated execution of the load tests, and (4) the automated regression detection.

In this chapter, the related works of the remaining parts are discussed. First of all, the related works about the workload determination for different system APIs are described in Section 3.1. Section 3.2 shows different declarative performance DSLs. In Section 3.3, a tool is described which supports the automation of performance tests. The Section 3.4 presents related works for the performance regression testing.

3.1. Regression Workload Determination

One major problem of testing services is that the interface can change. Therefore, the workload has to be adapted. One possible solution is to use the same workload for all service versions [FJA+10; SHNF15] with the problem that the workload does not always match to all versions. The reason for using the same workload is the automation and the assumption that an interface from a service does not change very often. Another way to detect anomalies, i.e., regressions, is to use load-independent approaches like Transaction Profiles (TP) described by Ghaith et al. [GWP+16; GWPM13]. A TP is derived from the results of a performance regression test and a queuing network model of the SUT. Compared to this research, the discussed approach requires a queuing network model before testing which is not available in this research.

3.2. Declarative Performance DSL

A declarative performance DSL is a part of the vision described by Walter et al. [WHK+16]. They describe a vision of *Declarative Performance Engineering* (DPE) which solves the problem that applications of performance engineering techniques are challenging and not executable without expert knowledge. The aim of DPE is to decouple *what* user concerns exist, e.g., questions and goals, which should be separated from *how* the concern should be solved, i.e., the technical part of a specific solution approach. The approach consists of (1) a declarative language where the user can specify what she wants to do, e.g., defining goals, (2) generic adapters which process the declarative language and execute different evaluation approaches, and (3) a decision engine which selects the optimal adapter for the execution.

A realization of the vision of DPE is the *Descartes Query Language* (DQL) presented by Gorsler et al. [GBK14]. DQL is for defining performance queries to predict performance metrics. It provides an Application Programming Interface (API) for a set of interfaces from performance modeling formalisms and their corresponding prediction techniques. DQL allows to define performance queries using modeling formalisms without knowledge about the definition of the formalisms because DQL works as a Wrapper for the included formalisms and hides the details. The benefit is that the manual effort using modeling formalisms is reduced caused by the common API. The DQL defines performance metrics, constraints, and goals for the performance prediction. An extension of the DQL is described by Walter et al. [WOK17]. The extension includes service level objectives (SLO) and service level agreements (SLA) definitions with the corresponding evaluation. To automatically answer the user concerns described by the DQL, tools are required which are demonstrated by Walter et al. [WEG+18]. These tools work with measurements from Kieker [VWH12] by a DQL adapter for Kieker [BPV+16] and software performance models or more precisely architectural performance models.

Another domain-specific language and automated toolchain iDSL is introduced by Van den Berg et al. [VHH18; VRH14]. In the iDSL language, the user concerns are separated from the specific solution approach [WHK+16], i.e., iDSL is declarative. The purpose of iDSL is the performance evaluation of service-oriented systems with model checking and simulations, e.g., selecting a design alternative can be supported. The performance analysis is executed automatically, and the results can be visualized with iDSL.

A goal of this research is to provide a declarative performance DSL according to the vision of Walter et al. [WHK+16]. The limited number of available declarative performance DSLs are caused by the early state of the declarative research. For other domains, more declarative DSLs are available as the survey from Van Deursen et al. [VKV00] shows. The available DSLs do not allow to define workloads and regression-specific properties. Therefore, an own DSL is required.

3.3. Performance Testing Tools

Taurus [Bla18] is an open-source tool for the automation of performance tests. It allows to execute load tests with different open source load testing tools to provide the possibilities from all tools and to eliminate restrictions from individual tools. It is a framework which works as a wrapper for different load testing tools with the benefit that users do not have to learn and define load tests in the specific tool format. Instead, Taurus provides a DSL in a YAML format with the possibility to define load tests in a readable form. Taurus supports the load testing tools Apache JMeter (see Section 2.4.4), Selenium[1], Gatling[2], "The Grinder"[3], and Locust.io[4]. The integration into a Continuous Integration (CI) server is possible, e.g., Jenkins[5], to automate the execution. Also, a command line interface exists for the manual execution of tests. In Taurus, the definition of criteria for passed and failed tests are available, e.g., an upper limit for the average response time. In contrast to BenchFlow, Taurus cannot deploy the SUT, e.g., a Docker-based application, and it does not support the definition of regression conditions based on previous SUT versions.

3.4. Performance Regression Testing

The manual execution of performance regression tests is time-consuming and error-prone [FJA+10]. Therefore, an automated execution is required. The execution of regression testing can be divided into the phases (1) test selection, (2) regression detection, and (3) root cause detection [JWS+18]. The first phase aims to solve the problem of an increasing number of tests which requires more execution time than available if systems are released frequently. Possible solutions are minimizing the number of tests [YH12], selection [KJMG17; YH12], and prioritization the tests [RUCH01; YH12]. We tackle a different problem in this research but the approaches can be combined with this approach.

The second phase comprises the execution of the regression tests to detect regressions. Performance regression testing can be done on various levels, e.g., on the highest level or the system level [JHHF09], on the service level or the component level with an interface

[1]Selenium: https://www.seleniumhq.org/
[2]Gatling: https://gatling.io/
[3]The Grinder: http://grinder.sourceforge.net/
[4]Locust.io: https://locust.io/
[5]Jenkins: https://jenkins.io/

[BK17; WW17], or on the class level [PHG14]. The approach of this research can be used on the system and service level.

For the comparison of the performance of an application version with one or more previous versions, a baseline is required. The baseline can be calculated from a performance regression testing repository which stores all previous test result reports [FJA+10] or from another storage which contains the performance test results, i.e., performance measurements [JHHF09; WW17]. Brunnert et al. [BK17] describe an approach which can only compare two versions based on resource profiles whereby the previous one has to be stored. Another approach from Pradel et al. [PHG14] does not require to store performance data, because the test generator requires two versions as input to generate the test. Compared to the approach from this research, the performance data is stored, but not in all cases it is possible to reuse the data caused by changing workloads. However, solutions exist for performance regression testing which are workload-independent [GWP+16; GWPM13].

In the third and last phase, the results of a performance regression test can be used for an automated root cause detection which reduces the manual interventions and the time for finding the root cause of a performance problem. Heger et al. [HHF13] realize this in an approach which is placed in the continuous development using continuous integration. The approach uses unit tests and a revision history graph to find the performance root cause. Lee et al. [LCL12] present a framework to automatically detect performance anomalies in the DBMS development using statistical process control charts and load tests. Also, Shang et al. [SHNF15] introduce an approach for automated performance regression detection based on clustered performance counters to build corresponding regression models. Performance counters are recorded from an old and a new application version using the same workload. The aim of this research is not to find the root cause, only to detect a possible performance regression.

Chapter 4

Approach

This chapter explains an approach to detect regressions based on the results of load tests. The execution of the approach is configured with a declarative DSL, and the approach can be used in a continuous software engineering environment.

For the approach, several tools are used. BenchFlow is a platform for end-to-end automation of performance testing with a declarative DSL for defining load tests, their goals, and their deployment. However, BenchFlow does not support HTTP workloads and their execution as well as the definition of regression conditions and their analysis. Therefore, we need to extend the BenchFlow DSL to support HTTP workloads and regression conditions. Also, BenchFlow should support to execute load tests with JMeter and to trigger a regression analysis for detecting regressions in load test results of several SUT versions. To complete the approach, a tool for continuous software engineering is required. ContinuITy supports continuous software engineering with a representative workload based on monitored user actions. The problem is that ContinuITy and BenchFlow cannot interact with each other because both tools use another format to store workloads. Therefore, a transformation between both tools is required.

This chapter is structured in the following sections. First of all, Section 4.1 describes the general overview of the approach of this thesis. The BenchFlow DSL extensions are explained in Section 4.2. The creation of the BenchFlow DSL in ContinuITy is covered by Section 4.3. The execution of the BenchFlow, including the execution with JMeter and the regression analysis, is explained in Section 4.4.

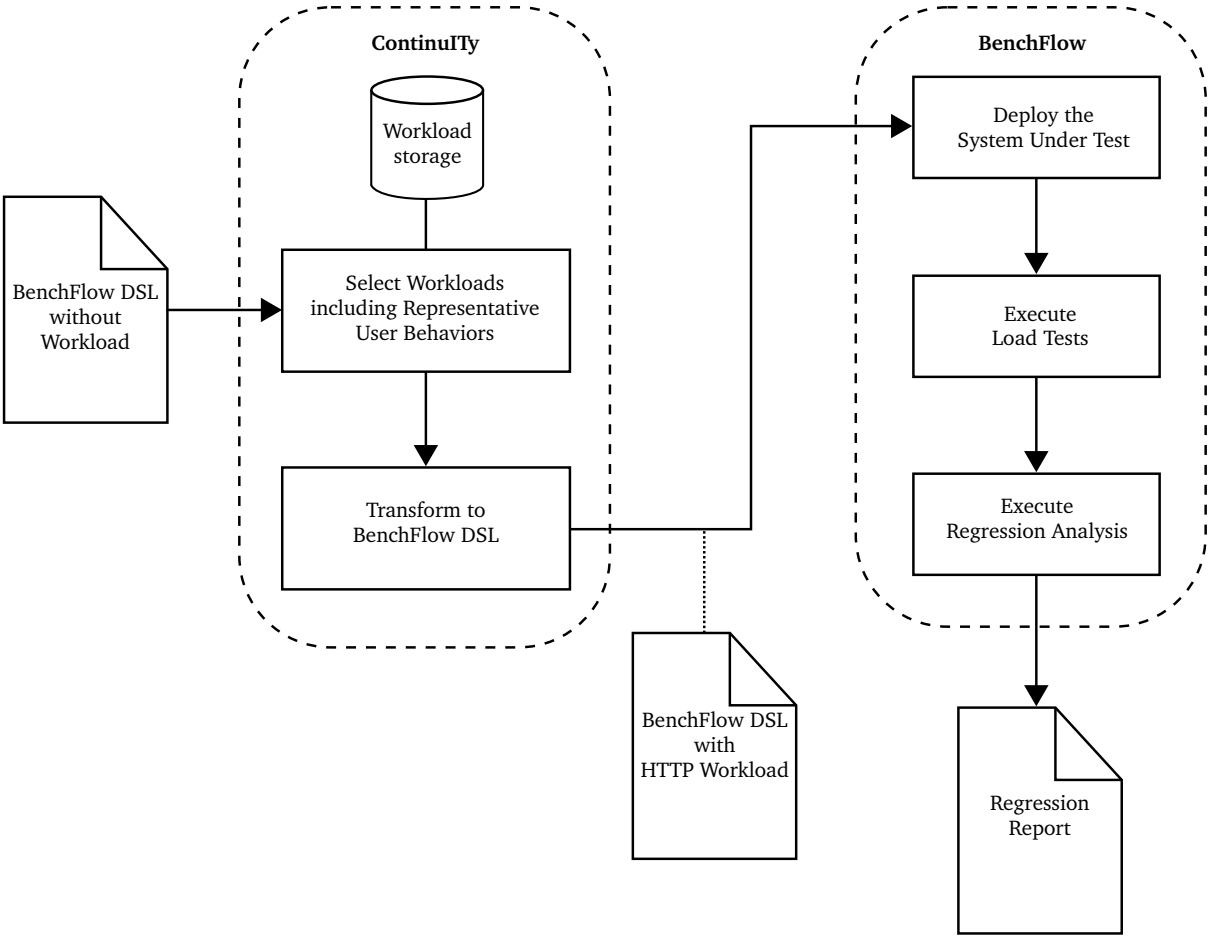


Figure 4.1.: Overview of the thesis approach.

4.1. General Overview

Figure 4.1 shows an overview of the thesis approach. The approach is divided into three different parts. The first part is the extension of the BenchFlow DSL to support HTTP workloads, regression conditions, and deployment properties. The extension is necessary for the next two parts which are the workflows of BenchFlow and ContinuITy.

A client configures a regression test using the BenchFlow DSL. Therefore, the SUT name and the SUT versions, which should be tested, are a part of the BenchFlow DSL. This BenchFlow DSL does not contain workload. The client provides the BenchFlow DSL to ContinuITy. This starts the workflow of ContinuITy. ContinuITy selects the suitable workloads for the defined SUT versions from a workload storage. This workload storage was already existing. These workloads include representative user behaviors. The selected workloads are transformed into a BenchFlow workload model, which is used for the provided BenchFlow DSL.

The BenchFlow DSL with the HTTP workloads is used for the workflow from BenchFlow. BenchFlow deploys the systems with the defined deployment properties. After deploying, load tests based on the HTTP workloads are executed on each SUT version. The performance results of each load test are compared with a regression analysis to detected regressions. Finally, the results of the regression analysis are visualized with a regression report.

During the two workflows of BenchFlow and ContinuITy, three model-to-model transformations are used to transform the models from ContinuITy into an executable load test. Figure 4.2 depicts these transformations. The workflow of ContinuITy contains the first transformation which transforms a WESSBAS model into a behavior model. This transformation is required to maintain the structure of the ContinuITy project.

Also, the second transformation is a part of the ContinuITy workflow. A BenchFlow DSL without workload defines the used SUT versions. For each version, there are three corresponding models which are the input models of the transformation. The input models are two IDPA models and the behavior model from the previous transformation. The result of the transformation is a BenchFlow workload model, which is a part of the BenchFlow DSL. The transformed workload models are appended to the provided BenchFlow DSL.

The BenchFlow workflow uses the completed BenchFlow DSL. This workflow contains the third and last transformation which is the transformation of the BenchFlow model into the JMeter test plan so that JMeter can execute load tests.

4.2. BenchFlow DSL Extension

BenchFlow supports HTTP workloads in a proof of concept version, which was a part of the work from D'Avico [DAv16]. The definition of HTTP workloads is not a part of the current BenchFlow DSL, and since the structure of the BenchFlow DSL changed, reuse is not possible. Also, the proof of concept version contains only a few definitions which are not enough for the approach proposed in this thesis. Available definitions for HTTP operations are (1) the endpoint URL, (2) the HTTP method, (3) request headers, and (4) the data for the body payload, but the definition of the protocol and other parameters, e.g., request parameter, is currently not supported. Therefore, the HTTP workload of the BenchFlow DSL is newly developed in this research. Also, BenchFlow does not support the definition of regression-specific properties which are necessary for this approach.

This section describes all necessary BenchFlow DSL extensions. To support HTTP workloads, the BenchFlow DSL is extended in Section 4.2.1. For the execution of the regression analysis, properties are necessary which define the regression goals

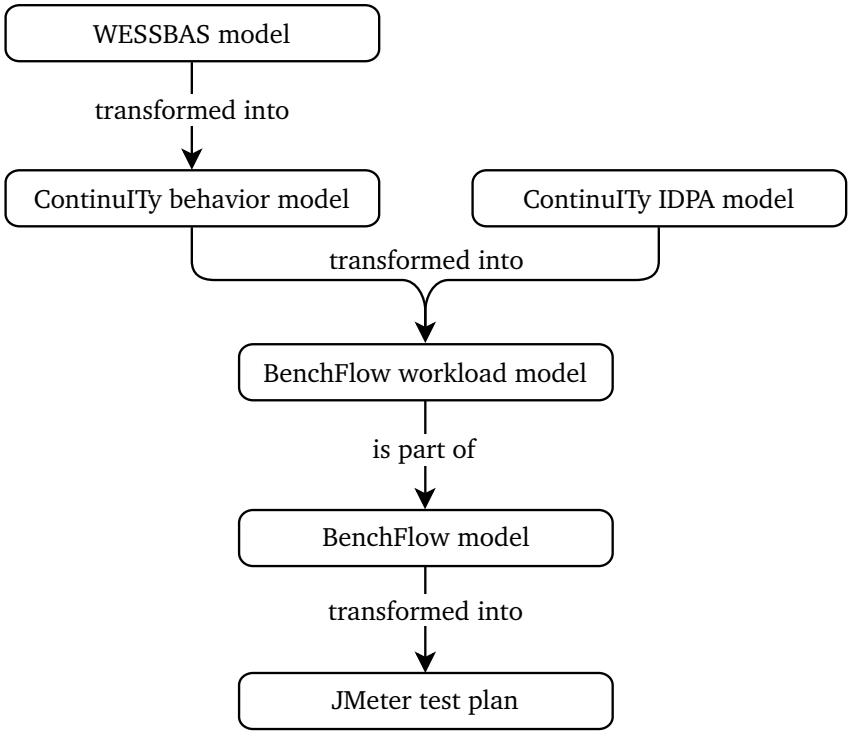


Figure 4.2.: Order of the model-to-model transformations.

Listing 4.1 An example of the structure of the BenchFlow DSL.

```
version: '1.0'
name: 'my-application'
sut:
  ...
configuration:
  ...
data_collection:
  ...
workload:
  ...
```

and regression conditions. This extension is described in Section 4.2.2. Section 4.2.3 describes the deployment properties which are required for the execution of BenchFlow. A complete BenchFlow DSL with all extensions is shown in Appendix A.3.

4.2.1. HTTP Workload Extension

This section describes the HTTP workload extensions of the BenchFlow DSL. Some HTTP workload extensions are inspired by Taurus [Bla18] because it allows to define HTTP

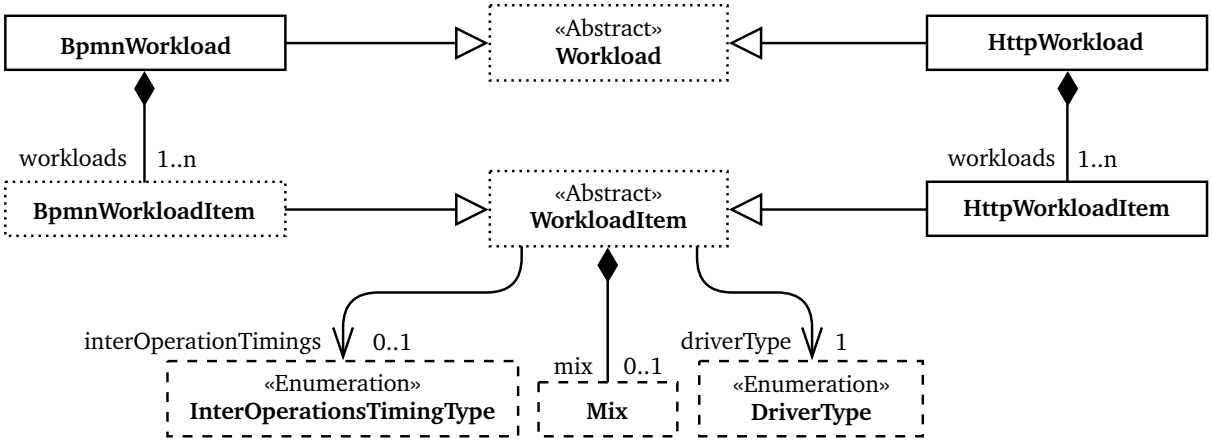


Figure 4.3.: Structure of the HTTP and BPMN workloads and workload-items. The classes which the dashed borders were already existing, and the classes with the dotted borders were a single class.

workloads in a YAML file like the BenchFlow DSL. The workload is defined on the root level of the BenchFlow DSL as visualized in Listing 4.1.

BenchFlow already supports the definition of Business Process Model and Notation (BPMN) workloads. A BPMN workload consists of BPMN models which are BPMN Workflow Management System versions. These versions can be compared with BenchFlow [Fin17]. To keep supporting BPMN workloads and to support HTTP workloads, an adaption of the current DSL structure is necessary. The structure of the workload is shown in Figure 4.3. The classes with the dashed borders were already existing, and the classes with the dotted borders were a single class that supported only BPMN. The structure of the BenchFlow DSL changes so that the user can use either BPMN or HTTP workloads. The BPMN workload contains a map of BPMN workload items, each with a list of files containing a BPMN diagram, and a set of workload item properties. The workload item properties define the common properties of the BPMN and HTTP workload items. With the HTTP workload, it is possible to define (1) the SUT versions, which are required for regression testing, (2) the global HTTP operations, and (3) the global data sources which can be used in the nested HTTP workload items. These two global definitions are only designed and not implemented in this thesis.

A data source represents a CSV file. An example with two data sources is depicted in Listing 4.2. A data source consists of (1) a path to a CSV file, (2) the delimiter of the CSV file, (3) an optional name, and (4) the row retrieval. The rows inside the file can be gone through sequential or at random. In the example, the first data source uses the default retrieval which is random, and the second data source explicitly defines the random retrieval. The limitation of this approach is that only random is supported and

Listing 4.2 An example of the definition of data sources in the BenchFlow DSL.

```
data-sources:
  - path: data.csv
    delimiter: ;
  - path: input.csv
    delimiter: ;
    retrieval: RANDOM
    name: input
```

Listing 4.3 An example of a matrix mix definition.

```
matrix:
- [ 20%, 80%, 0% ]
- [ 0%, 0%, 100% ]
- [ 0%, 40%, 60% ]
```

not sequential. The sequential retrieval is only designed and not implemented in this thesis.

Also, the HTTP workload item supports the definition of HTTP operations and data sources, but these can only be used inside the corresponding HTTP workload item. As mentioned before, the workload item properties are available for the HTTP workload items. The properties consist of (1) the probability that this workload item is selected. (2) The driver type which is either *START* or *HTTP*. *START* is used in the context with BPMN workloads and *HTTP* with HTTP workloads [Fin17]. (3) The inter-operation time type which is not used in this approach but it is used in the context with BPMN workloads [Fin17], and (4) a mix which defines the calling order of the operations. The mix contains one mix type, whereby BenchFlow provides four different mix types. The mix types differ from the possibility to define the orders. In this research, only the matrix mix is supported because it supports the definition of Markov chains, which are required from WESSBAS and ContinuITy. The other mix types do not support the definition of Markov chains. The matrix mix consists of a square matrix with all operations defined in the HTTP workload item. Each operation is a part of the column and the row definition. The order of operations in the matrix is the same as the order of the operations in the HTTP workload item. The row defines the current operation, and the column defines the possible next operation. An example of a matrix definition is depicted in Listing 4.3, whereby three operations are defined in the HTTP workload item. The first operation from the workload item has a probability of 20 % to call itself and 80 % to call the second operation. The second operation cannot call itself but only the third operation. The third operation can call itself with a probability of 60 %, and it can call the second operation with a probability of 40 %.

To support think times between operations, the transitions of the mix types are extended by the think time mean and the think time deviation in addition to the transition

Listing 4.4 An example of a matrix mix definition with think times.

```
matrix:
- [ 20% tt(140 45), 80%, 0% ]
- [ 0%, 0%, 100% tt(240.52 0) ]
- [ 0%, 40%, 60% ]
```

Listing 4.5 An example of an instance of a BenchFlow HTTP operation.

```
id: setStorageUsingPOST
endpoint: /storage/set/${id}
method: POST
protocol: http
headers:
  Content-Type: application/json
url-parameter:
  id: test
body:
  content:
   - Java
   - Scala
   - SQL
extract-regexp:
  extract_id:
    pattern: (.*)
    default: 'NO VALUE'
```

probability. This think time information is optional, while the probability is mandatory. An example of a matrix mix with think times is shown in Listing 4.4 which is an extension of the example from Listing 4.3. The first operation can call itself with a think time mean of 140 milliseconds and a think time deviation of 45 ms. The second operation can call the third operation with a think time of 240.52 ms. For the other transitions, no think times are defined.

The structure of HTTP operations is depicted in Figure 4.4, and an example of an operation instance is shown in Listing 4.5. The operation consists of

- an id for the identification,
- an endpoint for the path to an HTTP service endpoint,
- a list of HTTP headers for the request, e.g., *Content-Type*,
- an HTTP method, e.g., GET and POST,
- a protocol, HTTP or HTTPS, and
- body payload, parameters and response extractions.

The body payload can be either a file, a list of possible body strings, or a tuple of key-value-pairs for a body form. In addition to the body form, parameters can be used in the query string and in the URL, which are in both cases tuples of key-value-pairs. The value

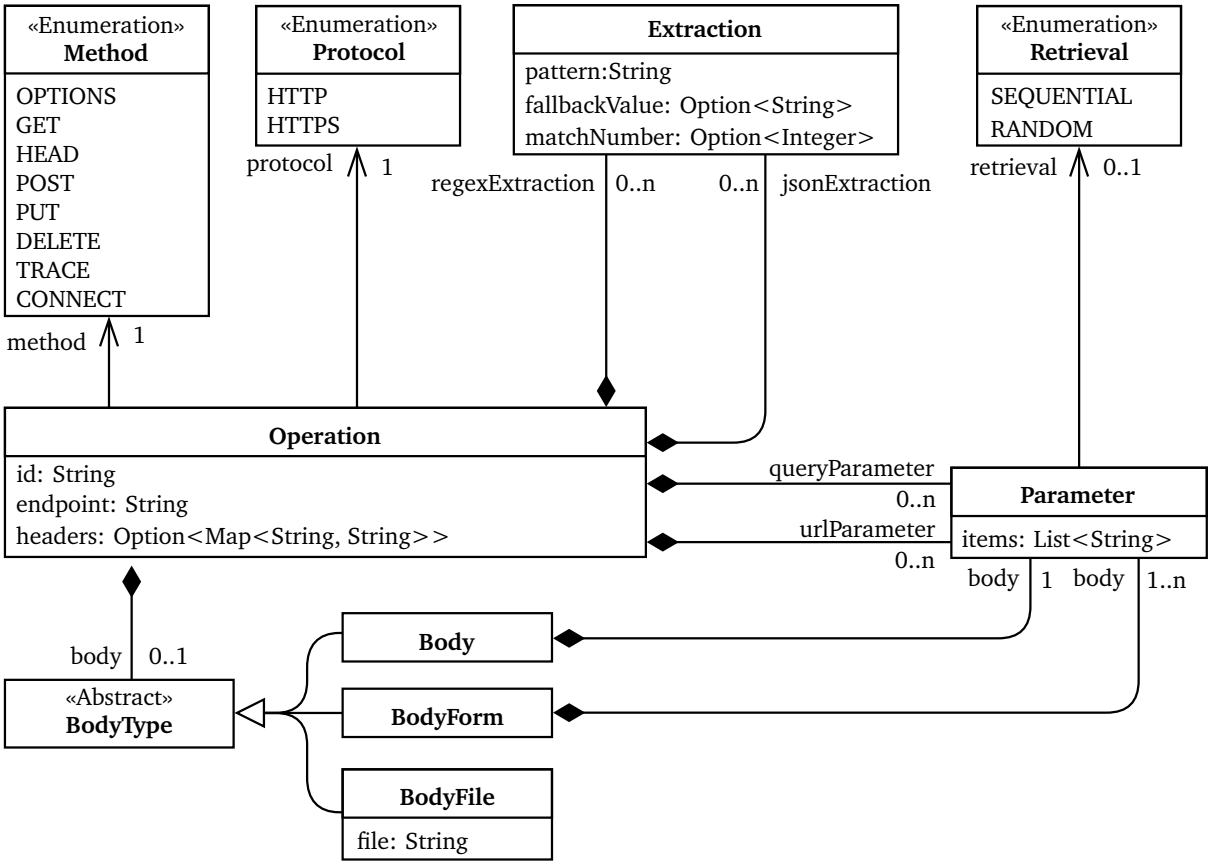


Figure 4.4.: Structure of HTTP operations in BenchFlow.

of a pair can be a list of possible input values, whereby the selection of the input value can be defined by the *Retrieval*. As in the data sources, the sequential retrieval is not supported. The response extractions of an operation can be either a regular expression or a JavaScript Object Notation (JSON) extraction, which work on the result of a request. The definition of an extraction is a list with key-value-pairs. The key represents the variable name which holds the extracted value. An extraction consists of (1) a pattern, (2) a default value when the pattern does not provide any results, and (3) a matching number when a certain result of a list of results should be selected.

4.2.2. Regression Extension

The execution of the regression analysis requires properties. These regression-specific properties are a part of this BenchFlow DSL extension. This extension includes regression goals and regression conditions. Section 4.2.2.1 explains the regression goals and Section 4.2.2.2 describes the regression conditions.

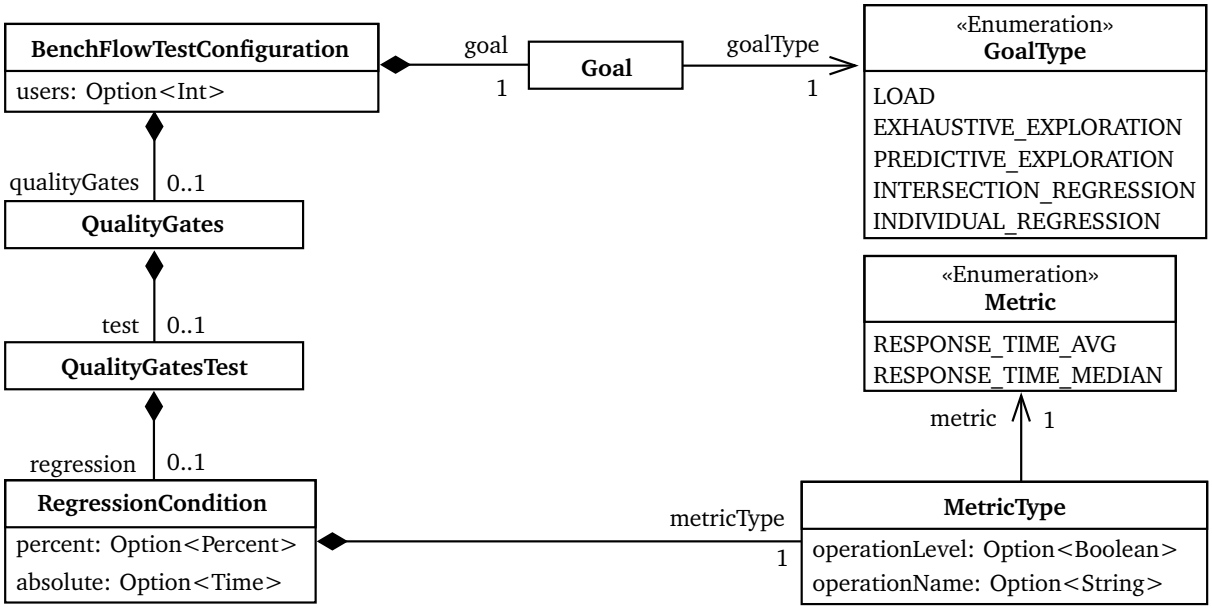


Figure 4.5.: Structure of the regression-specific properties in BenchFlow. This figure does not show all the attributes of the included classes, but only the attributes and classes that are important for the regression extension. Full class diagrams of the BenchFlow DSL are included in the work of Findahl [Fin17].

4.2.2.1. Regression Goals

Each BenchFlow DSL consists of a goal which defines what should be achieved with the underlying BenchFlow DSL. A goal consists of one of five goal types, two of which are the new types developed during this research. The new types supporting different regression methods are *INTERSECTION_REGRESSION* and *INDIVIDUAL_REGRESSION*. The regression methods are described in Section 4.3.5. With the *INTERSECTION_REGRESSION*, all defined SUT versions in the BenchFlow DSL are load tested with the same HTTP workloads and the same behaviors. The HTTP workload contains only HTTP operations which are available in all versions. For each version, the behavior is adjusted according to the new HTTP workload. For load testing, the versions use the behaviors from all versions. With the *INDIVIDUAL_REGRESSION*, each defined version in the BenchFlow DSL is load tested with the corresponding HTTP workloads and behaviors. The structure of the BenchFlow DSL including the goal type is displayed in Figure 4.5. The definition of a goal including the goal type is nested under the YAML key *configuration*. An example of the definition is shown in Listing 4.6.

Listing 4.6 An example of a goal definition inside a BenchFlow DSL part.

```
configuration:
  goal:
    type: INTERSECTION_REGRESSION
```

Listing 4.7 An example of a BenchFlow DSL part with a regression condition.

```
configuration:
  quality_gates:
    test:
    regression:
      metric: avg-response-time@@operation
      delta_absolute: 1ms
      delta_percent: 10%
```

4.2.2.2. Regression Conditions

This section describes the BenchFlow DSL extension by a regression condition. The regression condition is nested under the YAML key *configuration*. An example of a regression condition is shown in Listing 4.7. The YAML structure of the regression condition is based on the design of the quality gates of BenchFlow. Ferme et al. [FP18] describe the design of the quality gates of BenchFlow. The definition of the quality gates in BenchFlow is not possible so far. Therefore, the quality gates extensions that are necessary for this research are part of it. The extensions of the quality gates include the elements *quality_gates* and *test*, and the extension of the regression conditions includes the *regression* element with the nested elements. The structure of the BenchFlow DSL including the regression condition is displayed in Figure 4.5. This figure does not show all the attributes of the included classes, but only the attributes and classes which are important for the regression extension. The current implementation of the approach supports only one regression condition. More conditions are part of the future work in Section 7.3.

The following settings are available for the regression condition:

- A metric which is based on numbers, e.g., response time. The available metrics are defined in an enumeration in BenchFlow, and each metric represents a metric from the JMeter results. Currently, avg-response-time and median-response-time are supported. For a comparison, the metric value defines the baseline.
- The absolute delta increases the baseline of the metric by a fixed value. The approach supports only time-based values because of the available metrics.
- The percent delta increases the baseline of the metric percentage.

If both deltas are defined, there are two baselines for the regression condition. If one baseline is exceeded, a regression is detected.

The regression analysis can be conducted on three different levels:

- The global level: Defines that a regression is detected if the metric value based on all measurements of a version increases compared to the metric value based on all measurements of another version. For example, the total average response time of all operations can be defined by *metric: avg-response-time*.
- The operation level: Defines that a regression is detected if one operation increases its metric value compared to the metric value of the same operation in another version. For example, the comparison of the average response time of each operation can be defined by *metric: avg-response-time@@operation*.
- The operation-specific level: Defines that a regression is detected if the operation with the defined ID increases the metric value compared to the same operation of another version. For example, the average response time of the operation with the ID *Login* can be defined by *metric: avg-response-time@Login*.

The approach only supports the definition of a single regression condition because more implementations of different definitions are out of the scope of this thesis. The limitations of this extension are: (1) that only one regression condition can be defined and not a set of conditions. (2) Only a single metric can be used in the condition and not a combined condition out of a set of metrics like in functions. In addition to that, (3) only metrics of operations can be defined and no metrics of other levels such as a service. Nevertheless, potential solutions are considered.

To solve the first limitation, a BenchFlow DSL structure is required which allows the definition of a set of regression conditions in a nested way. Listing 4.8 depicts a structure with nested *AND* and *OR*. Each can contain a list of regression conditions. In the example, a regression is detected if the first two conditions *A and B* in the *AND* are true and one of the conditions in the *OR*. The *OR* contains a nested *AND* with the conditions *C and D*. Both conditions have to be true, or the condition *E* has to be true. The boolean function to that is:

$$\text{regression} = A \wedge B \wedge ((C \wedge D) \vee E)$$

The second limitation is that combined conditions out of a set of metrics are not supported. To solve this limitation, an extension is necessary which is a new YAML key for the function. Therefore, arbitrary functions are possible. For example, function-based regression conditions can be defined as in Listing 4.9.

The third limitation is that metrics from different levels are not supported. To solve this limitation, a small extension is needed similar to the solution of the second limitation

Listing 4.8 An example of a BenchFlow DSL with a set of regression conditions. The <*REGRESSION CONDITION*> defines a placeholder for a regression condition.

```
regression:
  AND:
    conditions:
     - <REGRESSION CONDITION>
     - <REGRESSION CONDITION>
  OR:
    AND:
      conditions:
       - <REGRESSION CONDITION>
       - <REGRESSION CONDITION>
    conditions:
     - <REGRESSION CONDITION>
```

Listing 4.9 An example of a BenchFlow DSL with function-based regression conditions. The <*METRIC*> defines a placeholder for a metric.

```
regression:
 - function: (<METRIC> / <METRIC>)
 - function: (<METRIC> * 2) + (<METRIC> / 5)
```

The solution is a new YAML key for the required level. For example, a metric based on a service can be defined as depicted in Listing 4.10.

All three limitation solutions can be combined so that the user of a BenchFlow DSL has the choice of several regression condition definitions.

4.2.3. Deployment Properties

Multiple SUT versions are necessary for regression testing, but BenchFlow does not support multiple SUT versions. Therefore, an extension of the BenchFlow DSL was required. An SUT version is necessary to deploy a system because it defines the version of the used Docker image. The BenchFlow DSL already supports deployment properties, but only the YAML keywords of the properties are important so that the properties can be found automatically.

Listing 4.10 An example of a BenchFlow DSL with a service metric regression conditions. The <*METRIC*> and <*SERVICE*> define placeholders for a metric and a service.

```
regression:
  service: <SERVICE>
  metric: <METRIC>
```

Listing 4.11 An example of a BenchFlow DSL with deployment properties.

```
sut:
  name: test-service
  version:
   - 1.2.3
   - 1.2.4
  type: http
  configuration:
    target_service:
      name: test-service
      endpoint: 192.168.012.123:8080
    deployment:
      docker_image: test-service
      docker_image_port_binding: 8080:8080
      docker_host_port: '2375'
      ping_url_path: /swagger-ui.html
```

An example of the deployment properties is shown in Listing 4.11. The list of the SUT versions or rather Docker image versions are part of the parameter *version* which is nested under the YAML key *sut*. The other deployment properties are nested under the YAML key *configuration* which is nested under the YAML key *sut*. The *configuration* contains *target_service* and *deployment*, both parameters are used for the deployment. The target service contains the service name and the endpoint URL. The endpoint URL contains an IP address which is used for the remote deployment of the SUT. Docker is used for the deployment. The parameter *deployment* contains key-value-pairs of further properties. A mandatory field is the Docker image which defines the name of the Docker image which should be used for the deployment. The following fields are optional:

- The field of the Docker image port binding describes which port is used from the Docker image. No port is used by default.
- The field of the Docker host port describes the open Docker port of the target system. A Docker port is required to deploy a Docker-based application remotely. The default port is 2375.
- The field of the ping URL path is used to ping the SUT after executing the Docker image. This path is used to check whether the SUT is running so that the load test can be started. The URL of a ping consists of the target service endpoint and the defined path suffix. No path is used by default. In this case, only the target service endpoint is used as URL.

4.3. ContinuITy Workflow

In Figure 4.6, the pipeline for the creation of the BenchFlow DSL is presented. This pipeline is developed during this research and it is a part of the ContinuITy project. The input of the pipeline is a BenchFlow DSL without HTTP workloads. This missing HTTP workload is created during the execution of the pipeline so that the result of the pipeline is a complete BenchFlow DSL including the HTTP workload.

The first step (1) of the pipeline is to select all ContinuITy models which are defined in the BenchFlow DSL. A ContinuITy model is a tuple of three models which are described in Section 2.4.3. Each version has a corresponding ContinuITy model. (2) After selecting all models, the pipeline distinguishes between the used goal type. The goal type provides the possibility to achieve different goals with BenchFlow. If the goal type is *INTERSECTION_REGRESSION*, (3) an intersection based on the ContinuITy models is calculated. The result of the intersection is exactly one ContinuITy model. If the goal type is not *INTERSECTION_REGRESSION*, each defined SUT version consists of an own ContinuITy model. For example, if the goal type *LOAD* is used, which is not introduced in this research, only one version can be defined which leads to exactly one ContinuITy model. Another goal type is the *INDIVIDUAL_REGRESSION* which can contain any number of ContinuITy models. The ContinuITy models which are not part of the goal type *INTERSECTION_REGRESSION*, are used directly for the next step (4). Each ContinuITy model from the intersection or which is used directly is transformed into a BenchFlow workload model. (5) All BenchFlow workload models are appended to the provided BenchFlow DSL so that the BenchFlow DSL is completed.

Section 4.3.1 describes the behavior model which is used to manage the behavior of an SUT version. The transformation of the WESSBAS model into the behavior model is explained in Section 4.3.2. Furthermore, the behavior model is a part of the ContinuITy model. Section 4.3.3 describes the ContinuITy model. The transformation of the ContinuITy model into the BenchFlow workload model is explained in Section 4.3.4. Section 4.3.5 describes the regression methods consisting of the intersection and the individual method.

4.3.1. ContinuITy Behavior Model

The behavior model is an exchange format inside ContinuITy but can also be used outside of ContinuITy. It contains a list of probabilistic user sessions, and it is based on the WESSBAS model. The model is created by transforming a WESSBAS model. Strictly speaking, the behavior model is a subset of the WESSBAS model with the behavior mix, behavior models, and the Markov states. It contains only the information which is

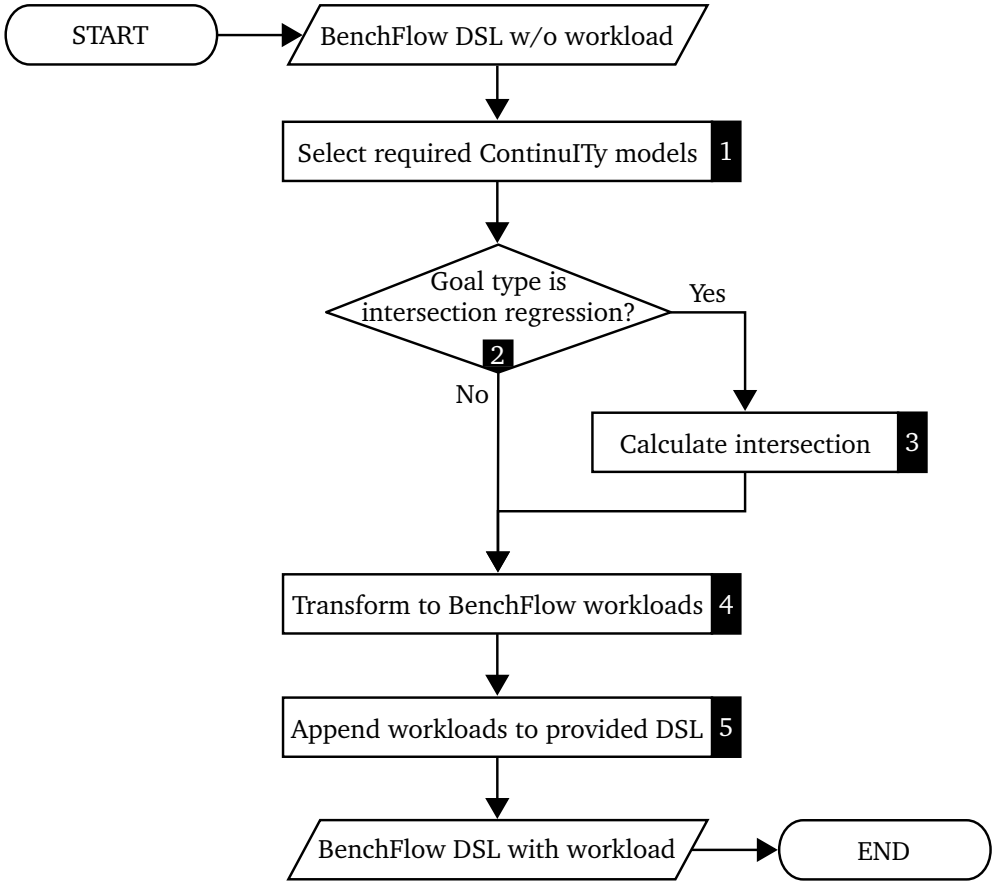


Figure 4.6.: Pipeline for the creation of the BenchFlow DSL.

required for the second transformation to create the BenchFlow model and which are not redundant in the IDPA models.

The behavior model consists of a set of behaviors, each contains (1) a name, (2) an initial state, (3) a probability of the user session, and (4) a set of Markov states. In contrast to the WESSBAS model, the probability of the user session is not separated from the behavior. Each Markov state contains a set of transitions to other Markov states. The transition consists of the (1) target state, (2) the transition probability, (3) the mean of the think time, and (4) the deviation of the think time. The think time information is placed in the corresponding transition because it reduces the complexity and the amount of data. An example instance of the behavior model is shown in Listing A.2.

4.3.2. Transformation from WESSBAS to Behavior Model

The transformation uses the WESSBAS model as input, and it provides the behavior model as result. This transformation uses the APIs of the WESSBAS and the behavior models, and it does not work on the individual forms. This transformation is required because the ContinuITy project is designed that only a single subproject knows the WESSBAS model. Therefore, the output model of the transformation is developed in this research. Since the transformation into the BenchFlow DSL needs only a subset of the information which is provided in the WESSBAS model, this transformation has the advantage that the required information can be filtered. Therefore, the next transformation has to handle only the required information and not the complete WESSBAS model.

4.3.3. ContinuITy Model

The ContinuITy model consists of the IDPA application, IDPA annotation, and the behavior model for a specific SUT version. The first step is to select the IDPA models, and the second step is to select the behavior model. The behavior model is a model which was developed during this research. The design of the model is based on the WESSBAS model but contains only a subset of information. This behavior model is not a native format of ContinuITy. Therefore, a transformation from the WESSBAS model into the behavior model is required. Selection of the behavior model means that in the first step, the WESSBAS model is selected. The second step is to transform the WESSBAS model into the behavior model.

It is possible that no WESSBAS model is available for a specific SUT version because no representative user behavior is created for a version so far. In this case, selecting the behavior model leads to an empty model. The pipeline for the creation of the BenchFlow DSL is stopped when for each version, a behavior model is required. This is the case for the goal types *LOAD* and *INDIVIDUAL_REGRESSION*. For the goal type *INTERSECTION_REGRESSION*, at least one behavior model is required out of all requested versions because the approach for this goal type merges only the available behavior models. The advantage of this is that a new version without existing representative user behavior can be load tested with regression analysis.

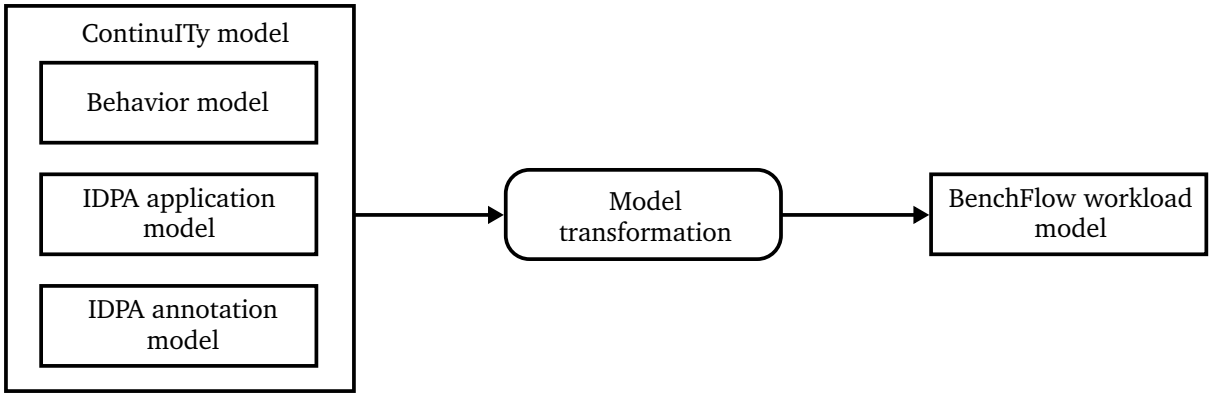


Figure 4.7.: Transformation of the ContinuITy model into the BenchFlow workload model.

4.3.4. Transformation from ContinuITy Model into BenchFlow Workload Model

Figure 4.7 shows this transformation. The input model of the transformation is the ContinuITy model, consisting of the behavior model, the IDPA application model, and the IDPA annotation model. The behavior model was generated from the previous transformation, which transforms the WESSBAS model into the behavior model. The result of the model transformation is the BenchFlow workload model. The BenchFlow workload model is a part of the BenchFlow DSL, but the workload model contains only HTTP operations and their behaviors. This transformation uses the APIs of the models, and it does not work on the individual formats.

The transformation works as follows:

- (1) The number of ContinuITy behaviors, which are defined in the behavior model, defines the number of HTTP workload items in the BenchFlow DSL. An HTTP workload item consists of the HTTP operations and their behavior. The steps 2 to 5 are conducted for each ContinuITy behavior.
- (2) The ContinuITy behavior is transformed into a BenchFlow behavior which is needed for the HTTP workload item.
- (3) The IDPA application and IDPA annotation models contain the parameters and properties of the HTTP operations. For the transformation, only the HTTP operations are needed, which are defined in the ContinuITy behavior. The advantage is that both IDPA models can be used with different behaviors, since the IDPA models can contain more HTTP operations than a behavior requires.

- (4) For each defined HTTP operation in the behavior, the parameters and properties of the HTTP operation are transformed into a BenchFlow HTTP operation.
- (5) All transformed BenchFlow HTTP operations of the current behavior are used for the BenchFlow HTTP workload item.
- (6) The complete BenchFlow workload model is composed of all transformed HTTP workload items.

4.3.5. Regression Methods

The thesis approach supports two different regression methods, namely the intersection and the individual method. The regression method depends on the selected goal type in the BenchFlow DSL.

The regression methods differ in the HTTP operations and in the user behaviors, which are used for load testing. With different regression methods, it is possible to achieve different goals based on changing system interfaces. An example of a changing interface is shown in Figure 4.8. In the example, three system versions are visualized, each version contains a different number of interface methods and in part different methods. In some cases, it is better to test the complete system interfaces from different system versions, each with the corresponding behavior. With it, it can be determined if the performance of the system interface is better than of another system interface. For example, the majority of the user behavior changes if a new web page is used or the main parts of the system are changed. The reason for that is that new functions are available, which are primarily used by the users. In other cases, it is better to compare only the interface methods which are available in all system versions. The benefit is that only the same interface methods are compared and thereby it is possible to prove if this part is responsible for regressions. For example, if the system interface and the logic behind the existing methods change, it should be determined if these methods act as before the change.

Therefore, two methods are required, the individual method which is described in Section 4.3.5.1 and the intersection method which is described in Section 4.3.5.2.

4.3.5.1. Individual Method

The individual method compares the system performance based on the individual behavior of each SUT version. This method gets a suitable workload model for each SUT version, e.g., five workload models for five versions, and stores them into a container. This container is the BenchFlow DSL. An example is shown in Figure 4.8 in which the

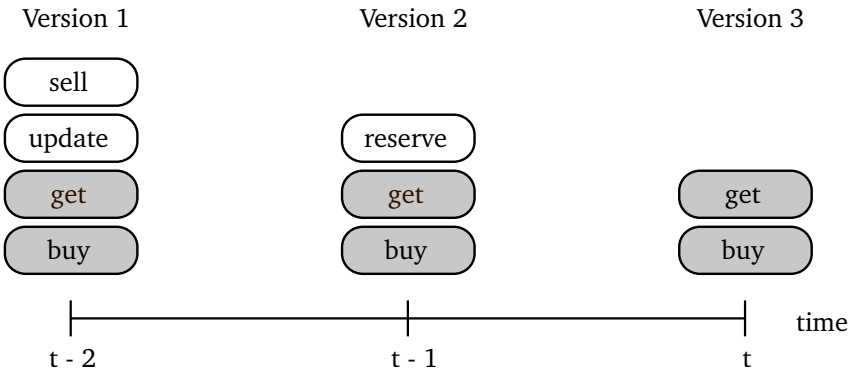


Figure 4.8.: Comparison of interface methods of different SUT versions over time. The methods in gray are available in all three versions.

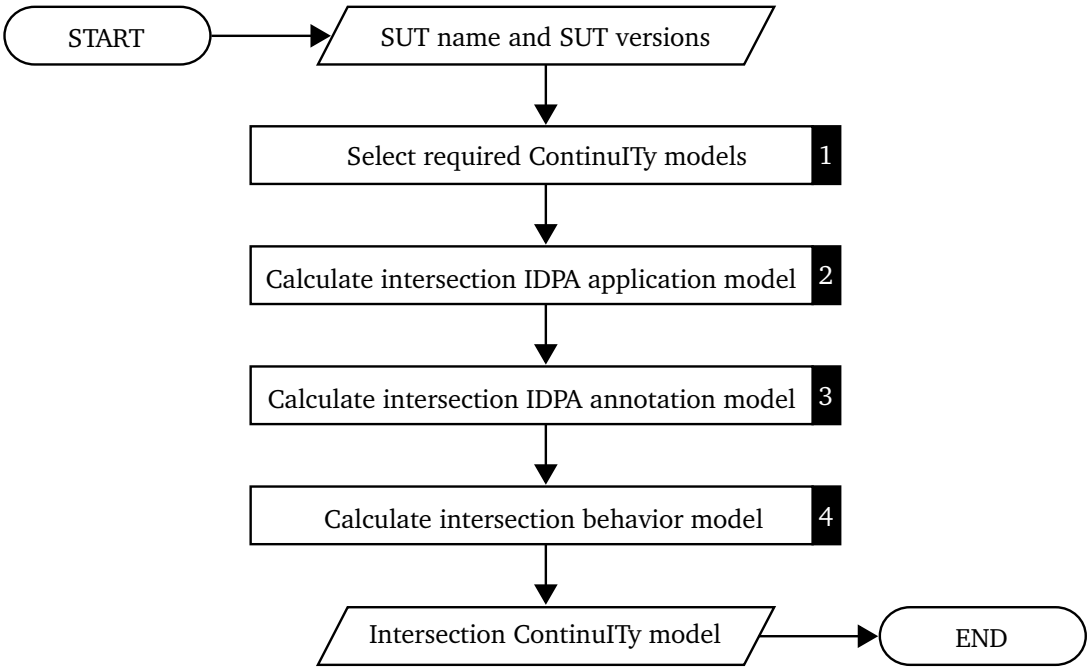


Figure 4.9.: Execution order of the intersection calculation.

workload model from version 1 contains four, version 2 contains three, and version 3 contains two interface methods. Two interface methods are available in all three versions. For each version, a separate load test is executed with its corresponding workload model which leads to measurement results for each version. A regression analysis compares the result measurements of all versions. The individual method can be used if the goal type of the BenchFlow DSL is *INDIVIDUAL_REGRESSION*.

4.3.5.2. Intersection Method

The intersection method uses only the interface methods which are available in all SUT versions for load testing. That enables to compare the same methods but with different SUT versions. In Figure 4.8, two HTTP methods are marked as gray elements which are available in all versions, these two methods are the intersection. The intersection method calculates one workload model out of a set of workload models, each representing an SUT version. The resulting workload model contains the intersection of the interface methods of each SUT version so that only the interface methods which are available in all versions are included.

ContinuITy executes the calculation of the intersection. Figure 4.9 depicts the execution order of the calculation. The calculation works as follows:

- (1) The first step is to select the ContinuITy model for each SUT version.
- (2) The next step detects changes in the IDPA application models. Changes are for example: new endpoints or new parameters. The changed interface methods are removed from the first IDPA application model so that one IDPA application model exists which represents the intersection of all other application models. In the example from Figure 4.8, two methods are part of the intersection of version 1 and 2.
- (3) In the next step, the intersection IDPA application model is used for the adjustment of the IDPA annotation model. All annotations that reference to deleted interface methods in the IDPA application model are removed. Also, extractions from deleted interface methods are removed.
- (4) An adaption of the interface methods requires changes of the Markov chain which is a part of the behavior model.

Figure 4.10 shows an example of how two Markov chains change during the intersection. The reason for the changes is that each Markov state represents an interface method. (4.1) All methods which are not available in the intersection have to be deleted in the Markov chain which is a part of the adaption of the Markov chain. Not all states are removed at the same time. The states are removed one by one because of the next step, namely the rebalancing, because the rebalancing only works for a single removed state. The state is removed after the rebalancing, but we call it removed state nevertheless. In the example, the corresponding interface method from state *B* was removed in the intersection. Therefore, state *B* is removed from version 2. After that, (4.2) the Markov chain has to be rebalanced so that all states have an outgoing probability of 100 %. The rebalancing works as follows: All states which have an outgoing transition to the removed state are used for the rebalancing. Anyway, after the

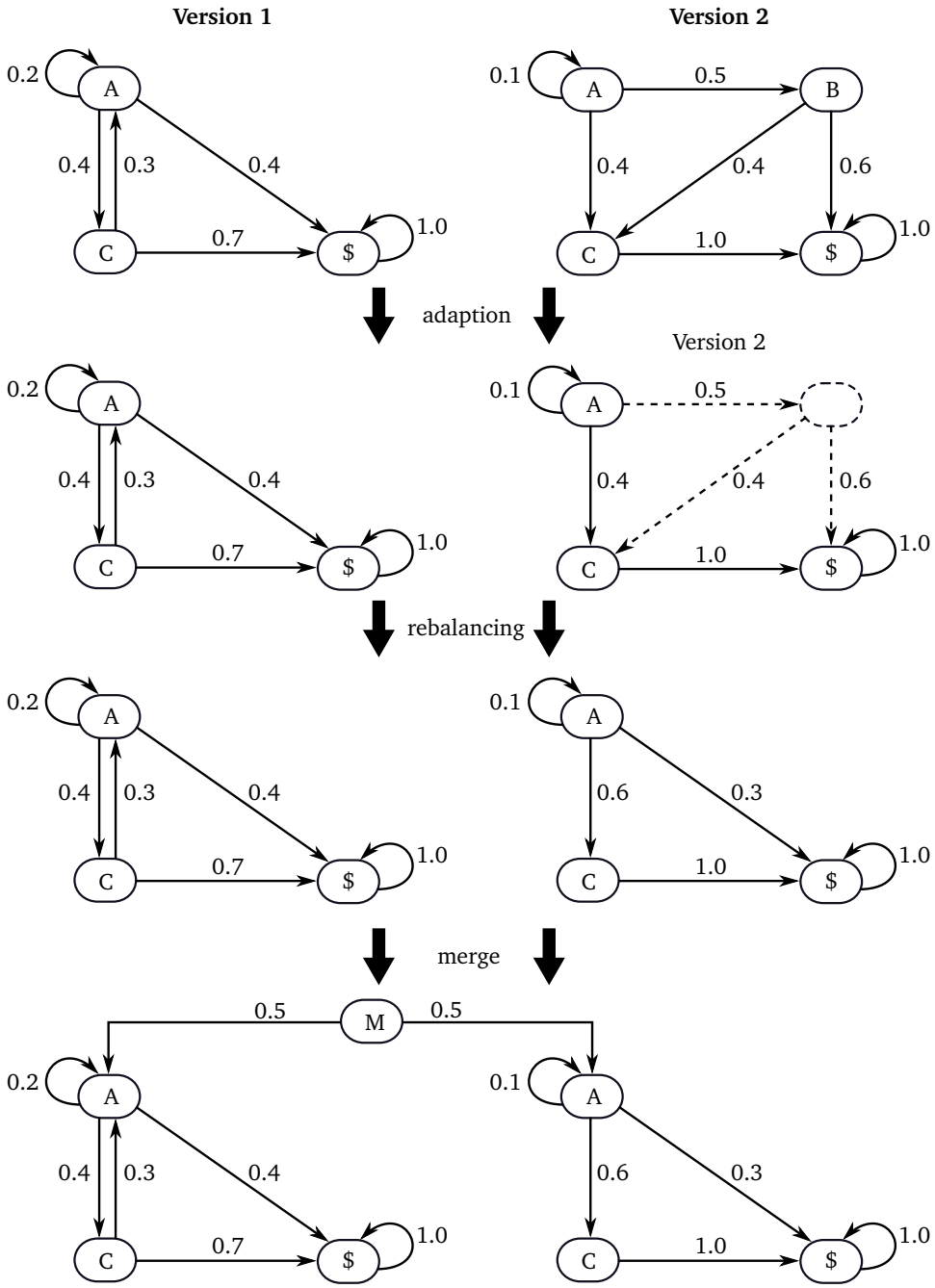


Figure 4.10.: Example of a Markov chain which changes by intersection. A, B, C, and M define Markov states and $ defines the final state of the chain. State B was removed by the intersection.

deletion, all other states have an outgoing probability of 100 %. If the removed state contains a transition to itself, the loop has to be removed otherwise the next step does not lead to an outgoing probability of 100 %. The loop probability is subtracted from the 100 % to reduce the overall outgoing probability of the removed state. For each other outgoing transition from the removed state, the probabilities are recalculated with the new overall probability. This recalculation leads to an increase of the probabilities so that the sum of all outgoing probability is 100 % without the loop transition. After removing the loop transition, each rebalanced state gets all outgoing transitions of the removed state but not with the same probability. For each transition, the new probability is calculated by the probability of the outgoing transition to the removed state multiplied with the old probability of the transition. If a rebalanced state contains two outgoing transitions to the same state, the transitions are merged that means the probabilities are summed. Finally, the removed state is indeed deleted otherwise the previous steps were not possible because the outgoing transitions of the removed state are a part of the removed state. In the example, a rebalancing of the Markov chain for version 2 is required because the transition *A* to *B* is deleted, which leads to that state *A* has only an overall outgoing probability of 50 % left. Therefore, the 50 % of the outgoing transition to the removed state is multiplied with 40 % of the transition to *C* which leads to 20 %. Since state *A* already contains a transition to state *C* the probabilities are summed. Also, 50 % is multiplied with 60 % of the transition to the end state which leads to 30 %. After rebalancing the state *A* has an outgoing probability of 100 % and a transition to the end state because before rebalancing a transition from *A* to *$* via *B* was possible.

For each available ContinuITy model, an adaption and rebalancing are required if a behavior model exists. In this regression method, a behavior model is not necessary for all ContinuITy models because only the available behavior models are merged so that in the end, one intersection behavior model exists. The next step conducts the merge. The benefit of a not required behavior model is that when an SUT version has no representative user behavior, it can be load and regression tested nevertheless. The merging step is performed by appending each behavior or rather Markov chain to a single behavior model. Thereby, each behavior gets the same probability. In the example, the Markov chains from the versions 1 and 2 are merged by inserting a new state *M* with two transitions, and then connecting the new state with both starting states. Both transitions contain the identical probability of 50 %. This single behavior model is the intersection behavior model. All three intersection models consist of the IDPA application, IDPA annotation, and the behavior model form the intersection ContinuITy model for these versions.

As in the individual method, a separate load test is executed for each version but with the same workload model. The result measurements of all versions are analyzed by a regression analysis. The individual method can be used if the goal type of the BenchFlow DSL is *INTERSECTION_REGRESSION*.

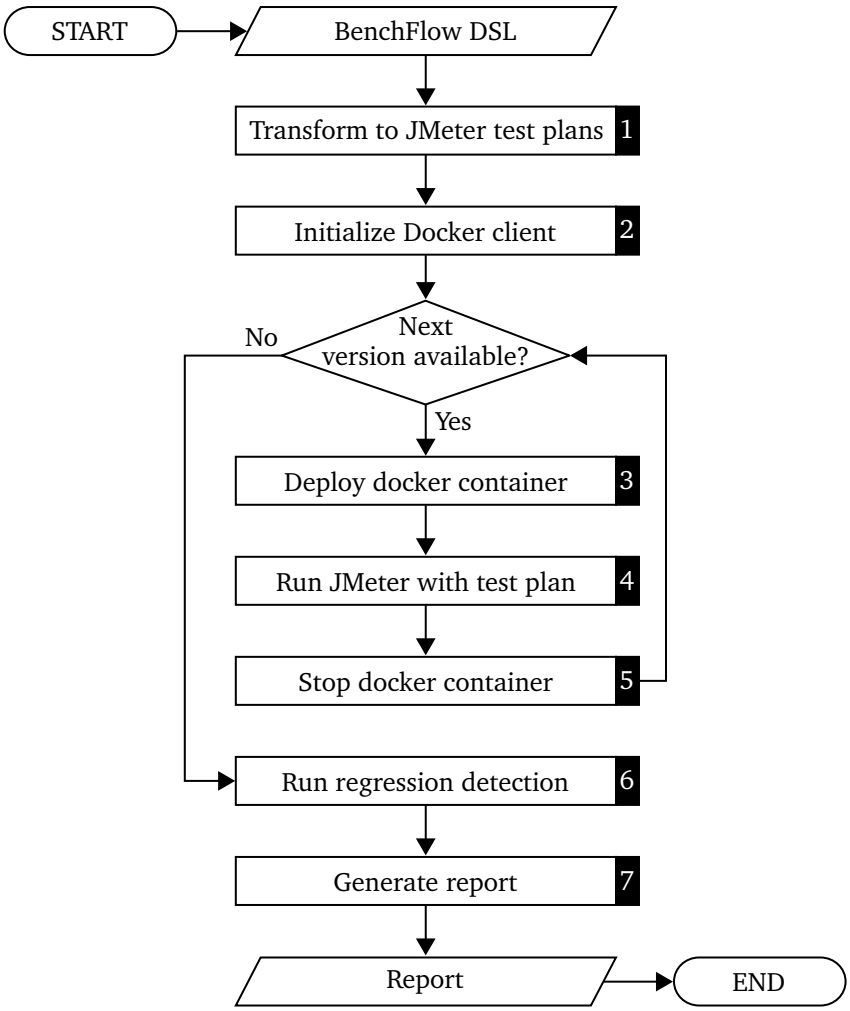


Figure 4.11.: Pipeline of the execution of a BenchFlow DSL using regression detection.

4.4. BenchFlow Workflow

The pipeline of BenchFlow using the regression analysis is depicted in Figure 4.11. The first step (1) is to transform the provided BenchFlow DSL into JMeter test plans. The number of provided test plans depends on the number of specified workloads in the BenchFlow DSL. The second step (2) is to initialize a connection to the test system with the deployment properties. The next step (3), the SUT is deployed with a Docker container based on the Docker image consisting of the SUT name and the SUT version. The Docker image is received from Docker Hub[1], if the image is not available on the test system. After deploying the SUT, a defined URL is pinged until the SUT is running.

[1]Docker Hub: https://hub.docker.com/

If multiple test plans are available because of multiple definitions of workloads, then the test plan is selected which matches the SUT version which is currently running. If only one test plan is available, then this test plan is used for all SUT versions and thus for all load tests. The number of test plans depends on the selected regression method. The next step (4) is to execute the selected test plan with JMeter which provides the results in a CSV file. After load testing this SUT version, the Docker container is stopped and removed (5). The steps 3 to 5 are repeated until the load tests for all defined SUT versions are executed. After executing all load tests, a regression analysis is triggered (6) which detects regressions based on the JMeter results and the specified regression conditions in the BenchFlow DSL. The results of the regression analysis are used in the next step (7) to create the regression report. The report provides all results in a visualized manner.

The following sections are structured as follows: Section 4.4.1 explains the transformation. The regression analysis is described in Section 4.4.2 and the regression report is explained in Section 4.4.3. In Section 4.4.4, the files and results are mentioned, which are stored in a single directory.

4.4.1. Transformation from BenchFlow DSL to JMeter Test Plan

The transformation uses the BenchFlow DSL, including the BenchFlow workload model, as input and provides the JMeter test plan as result. The BenchFlow workload model can be created by the previous transformation, see Section 4.3.4. The transformation is required because a test plan is necessary to execute load tests with JMeter. The test plan is the configuration file for executing load tests with JMeter. JMeter does not support probabilistic user behaviors by default. Therefore, the JMeter extension *Markov4JMeter*[2] is used which provides the possibility to define Markov chains including think times for each transition.

This transformation uses mainly the BenchFlow workload model and other properties which are nested under the YAML key *configuration*, an example is shown in Listing 4.12. The BenchFlow DSL supports already the necessary properties, and therefore no extension was required. For the JMeter test plan, the following fields are necessary:

- the number of users,
- the ramp-up time which defines the duration in which all users are created, and
- the max time of the termination criteria for defining the execution time of the test.

[2]Markov4JMeter: https://github.com/Wessbas/wessbas.markov4jmeter

Listing 4.12 An example of a BenchFlow DSL with required fields for the JMeter transformation.

```
configuration:
  ...
  users: 2
  workload_execution:
    ramp_up: 0m
    steady_state: 10m
    ramp_down: 100m
  termination_criteria:
    test:
      max_time: 2s
    ...
```

For each workload defined in the BenchFlow DSL, an independent test plan is created by the transformation. That has the advantage that test plans for different SUT versions can be generated through a single BenchFlow DSL, which is necessary for the individual method.

4.4.2. Regression Analysis

The regression analysis detects regressions based on the measurements collected by JMeter and the specified regression conditions in the BenchFlow DSL. The regression analysis provides the detected regressions in a file so that other tools can evaluate and visualize the detected regressions. Section 4.4.3 explains the regression report which is the visualization of the detected regressions in this approach.

A regression is detected if the regression conditions of the BenchFlow DSL are fulfilled. The regression analysis works as follows:

- (1) Determine at what level the regression analysis works. Further steps are explained for the operation level because the other levels are similar.
- (2) For each SUT version, the monitored measurements from JMeter are grouped by the label name of the requests. The label name is equal to the operation ID of the BenchFlow DSL. The operation ID in the BenchFlow workload defines if the operation is the same or not the same as the operation from another workload. The same operation in different BenchFlow workloads has to contain the same operation ID. If this is not the case, the regression analysis considers each operation as an individual operation and since the analysis requires operation measurements of different versions, an analysis is not possible. The operation ID is unique within a workload.

Listing 4.13 Example of a regression condition for the regression analysis.

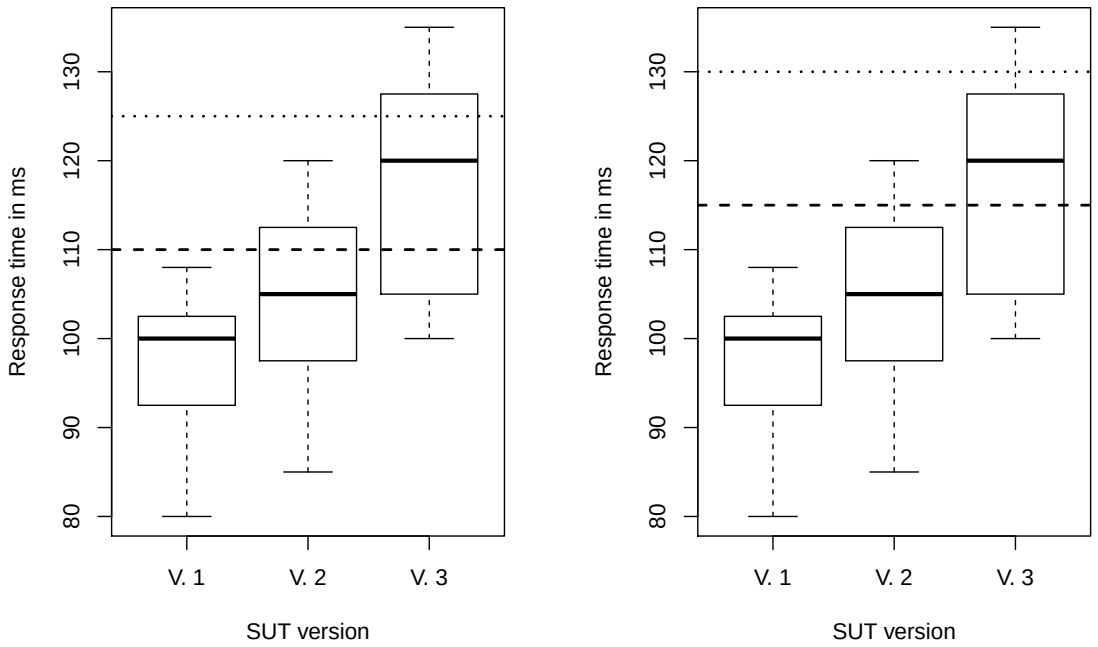
```
metric: median-response-time
delta_absolute: 25ms
delta_percent: 10%
```

- (3) For each operation, the corresponding measurements are aggregated by the defined regression condition metric which is a part of the BenchFlow DSL. This leads to a single aggregated measurement for each operation in each version.
- (4) For each operation ID, the aggregated measurements of all versions are merged. Therefore, for each operation ID, a list of versions with the corresponding aggregated measurements exists.
- (5) For each operation ID, the regression analysis compares the aggregated measurements of all versions with each other.

For each operation ID, the comparison works as follows:

- (5.1) The aggregated measurements of the operation are selected.
- (5.2) If more than one aggregated measurement exists, a comparison is possible. Otherwise, only one measurement of a particular version is available which cannot be compared with a measurement of another version.
- (5.3) A version is compared to all newer versions. For example, two versions lead to one comparison for each operation, and three versions lead to three comparisons for each operation. The aggregated measurement of the older version is used as the base for one or two baselines. The regression condition from the BenchFlow DSL allows to define two deltas, namely the absolute and the percent delta. If both deltas are defined, two baselines exist for the comparison. If one or no delta is defined, one baseline exists. In the case that no delta exists, the base forms the baseline. The usage of the absolute delta leads to a baseline of the base plus the absolute delta value. In the last case, the percent delta is used which leads to a baseline of the base plus the base multiplied by the percent delta value. If just one baseline is exceeded, a regression was detected.

For example, three SUT versions should be compared with the regression analysis. The regression condition is defined in Listing 4.13. Figure 4.12 shows two diagrams with the measurements of each version and different baselines. The dotted line is the baseline from the absolute delta, and the dashed line is the baseline from the percent delta. The baselines depending on the median of the first version are depicted in Figure 4.12a, and the baselines depending on the median of the second version are shown in Figure 4.12b. First of all, the regression analysis compares the oldest version with second oldest version, that means version 1 and 2. Version 2 has a median value of 105 ms, and



(a) Comparison with baselines from version 1. **(b)** Comparison with baselines from version 2.

Figure 4.12.: Example of a regression analysis.

the baselines from version 1 have the values 110 ms and 125 ms. Neither of them is exceeded. In the second comparison, version 1 is compared with version 3 with the same baselines. Version 3 has the median value of 120 ms which leads to that the baseline from the percent delta is exceeded. Therefore, a regression was detected. The third and last comparison is with the versions 2 and 3, and the baselines are depending on version 2. This leads to the baselines with the values 115.5 ms and 130 ms. Version 3 has a median value of 120 ms which leads to a regression because the baseline from the percent delta is exceeded. This example detects two regressions in three versions.

The advantage of this regression analysis is that a gradual increase in measurement values can be detected if more versions are compared because the newer versions are also compared to the older versions. The disadvantage of this regression analysis is that a regression can be detected which was resolved. For example, an intermediate version between the oldest and newest version can cause a regression because it performs worse than the oldest version but the newest version performs as well as the oldest version. However, a regression is detected because the intermediate version is compared to the oldest version.

The usage of another regression analysis level leads to minor differences in the performed steps. For the operation-specific level, the operations in step 2 are filtered by the

searched operation ID which is defined in the regression condition. The other steps are the same with the exception that exactly one operation has to be compared and not a set of operations. Using the global level, all operations in step 2 are merged without consideration to the operation ID. The other steps do not compare the operation measurements but global measurements, exactly one measurement for each version.

4.4.3. Regression Report

The regression report consists of (1) the detected regressions and (2) diagrams which visualize the measurements based on the regression condition metric. Since only the response time is supported as the metric, each HTTP operation contains a box plot and a density plot with the response time measurements. An example page is pictured in Appendix A.4 which contains both plots and a detected regression. The report generation is triggered at the end of the pipeline. It requires the monitored measurements from JMeter and the detected regressions from the regression analysis. The output of the report generation is a regression report as a PDF file. The regression report consists of:

- a title page with the SUT name and all detected regressions,
- an overview page with a box plot with the global response time for each SUT version, and
- for each operation, a single page with the detected regressions, a box plot and a density plot with response time measurements.

4.4.4. Workflow Results

All files and results which are needed and created during the execution of the BenchFlow workflow are stored in one directory. The directory consists of:

(1) the BenchFlow DSL in YAML format,
(2) for each SUT version, a JMeter test plan with the corresponding behavior files in CSV format,
(3) for each SUT version, the JMeter results as CSV and Java serialized file,
(4) the regression results with the detected regressions in JSON format, and
(5) the regression report with the detected regressions and diagrams as PDF file.

The benefit of the directory is that all intermediate results are stored and that these results can be used to rerun the workflow at a certain step so that the previous steps can be omitted. Therefore, it is possible to execute the load tests once and to use the JMeter results for several regression analyses with different regression conditions.

The regression analysis is explained in Section 4.4.2. Furthermore, the design of the regression report can be changed without executing the complete workflow again because the report generation from Section 4.4.3 can be executed with the intermediate results.

Chapter 5

Implementation

This chapter describes implementation parts of the approach. The structure of the BenchFlow DSL extensions were already mentioned in Section 4.2. The ContinuITy workflow is responsible for the HTTP workload including the representative user behavior. Section 5.1 describes the completion of the BenchFlow DSL with the involved services. The BenchFlow workflow is responsible for the load test execution and the regression detection. Section 5.2 describes implementation details about this workflow.

5.1. BenchFlow DSL Completion in ContinuITy

The BenchFlow DSL is created in this approach in ContinuITy (Section 4.3). Figure 5.1 shows the involved ContinuITy services.

The BenchFlow service was developed during this research in an own subproject inside the ContinuITy project. A caller requests the REST API of BenchFlow service with a BenchFlow DSL in YAML format which does not contain a workload. To create the BenchFlow DSL, the ContinuITy models of the provided SUT versions are required. A ContinuITy model consists of three different models which are managed by different services, namely the IDPA application, the IDPA annotation, and the Wessbas service. Each service consists of a REST API which supports Swagger[1]. For each SUT version, the three models are queried from the corresponding service so that a ContinuITy model can be created. Each ContinuITy model is transformed into a BenchFlow HTTP workload, and each workload is added to the provided BenchFlow DSL. The caller receives a response with the BenchFlow DSL consisting of the HTTP workloads.

[1]Swagger: https://swagger.io/

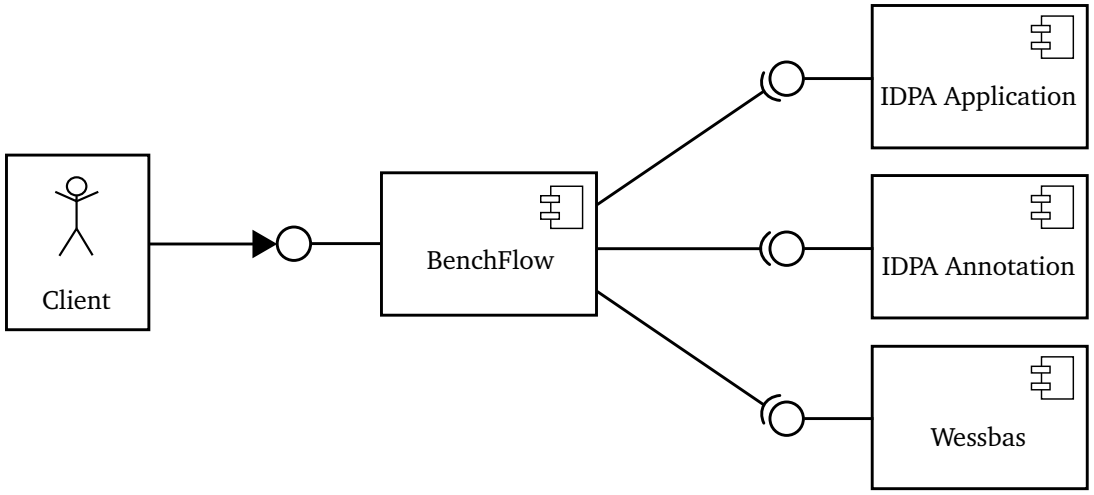


Figure 5.1.: Involved ContinuITy services for the completion of the BenchFlow DSL.

5.2. BenchFlow Workflow

The BenchFlow workflow is part of a standalone application. It is not integrated into the life cycle of BenchFlow. The project is implemented in Java and uses Gradle[2].

The execution of the BenchFlow workflow requires three information.

- The path to the BenchFlow DSL file.
- The path to the directory that should contain all workflow results. The workflow results are explained in Section 4.4.4.
- The path of the JMeter directory to execute load tests with JMeter. JMeter has to contain the extension *Markov4JMeter*[3] otherwise it is not possible to execute load tests with Markov chains.

It is also possible to integrate the BenchFlow workflow into another application. This application has the choice to use either a BenchFlow DSL file or directly a BenchFlow DSL object to execute the BenchFlow workflow. The project is designed that each step of the workflow can be executed alone. This has the benefit that the intermediate results, which are stored in the result directory, can be used several times, e.g., the JMeter results can be used for several regression analyses with different regression conditions. Therefore, time is saved.

[2]Gradle: https://gradle.org/
[3]Markov4JMeter: https://github.com/Wessbas/wessbas.markov4jmeter

Listing 5.1 An example of a JSON file containing a detected regression.

```
{
  "sut":"test-service",
  "regressions":[{
    "metric":"median-response-time",
    "operation_id":"get",
    "version_old":"test-service-2.1",
    "version_new":"test-service-2.2",
    "value_old":50.0,
    "value_new":150.0,
    "delta_absolute":1.0,
    "delta_relative":null
  }]
}
```

The regression analysis and the regression report are parts of the BenchFlow workflow. Implementation details about the regression analysis are described in Section 5.2.1 and about the regression report in Section 5.2.2.

5.2.1. Regression Analysis

The regression analysis is designed so that different regression algorithms can be used. One algorithm is developed during this research, and this is also the only one which is available. This algorithm is the default algorithm if no other algorithm is selected. A customized algorithm can be set before the BenchFlow workflow is executed.

An algorithm always receives the defined regression conditions from the BenchFlow DSL and the JMeter results. The structure of the algorithm results is always the same because the analysis returns these results. Furthermore, the same structure is required to generate a regression report otherwise the regression report cannot show the detected regressions correctly. When the results are stored, they are in JSON format. Listing 5.1 depicts an example of the results including a single detected regression.

5.2.2. Regression Report

The regression report (Section 4.4.3) visualizes the results of the regression analysis. The report is generated by the language R and can be triggered as a standalone application. It requires (1) CSV files containing the monitored measurements, which are in this approach the JMeter result files, and (2) a JSON file containing the detected regressions, which is provided from the regression analysis. The output is a regression report as a PDF file.

If the report generation is used in the pipeline, *Rscript* has to be a part of the environment. Otherwise, it is not possible to execute the report generation because the *Rscript* file is executed by command line automatically.

Chapter 6

Evaluation

The goal of this thesis is to provide an approach for automated regression testing based on load testing with representative user behaviors using a declarative approach. To evaluate if this main goal is reached, two different evaluations are conducted. The first evaluation is for the regression analysis which is explained in Section 6.1 and the second evaluation is for the extensions of the BenchFlow DSL which is described in Section 6.2.

6.1. Evaluation of the Regression Analysis

This section evaluates the regression analysis of the BenchFlow DSL. First of all, the research questions are explained in Section 6.1.1. After that, Section 6.1.2 envisions the experiment settings. The results of the experiment are described in Section 6.1.3 and in Section 6.1.4, the results of the scenarios are discussed. Section 6.1.5 describes the answers to the research questions. Finally, the threats of validity are mentioned in Section 6.1.6.

6.1.1. Research Questions

This section describes the research questions for the thesis approach.

RQ1: How well does the regression analysis detect regressions?

This first question aims to evaluate if performance regressions are correctly identified by the approach of this thesis. To specify this questions, two underlying questions exist which are the following both.

- **RQ1.1:** How many false negative and false positive regressions are detected?

  This question deals with how often regressions are detected which are no regressions and how often regressions are not detected which were regressions.

- **RQ1.2:** How well does the regression analysis detect regressions in SUT versions without existing real user behavior?

  This question aims to evaluate the continuous performance engineering part of this thesis to determine if the approach of this thesis can be integrated into an automated continuous integration pipeline. We want to investigate how the regression detection works for SUT versions without corresponding representative user behavior by executing at least one old SUT version with corresponding user behavior and a new SUT version without user behavior.

RQ2: What is the difference between the measured results of the individual and intersection methods?

In this thesis, two regressions methods are envisioned. This question investigates the difference between the two methods, the individual and the intersection method by comparing the measured performance results from JMeter. The following question extends this question.

- **RQ2.1:** Are different regressions detected with different regression condition levels?

  This question aims to evaluate if the different regression methods have impacts on the detected regression. This comparison covers two different regression condition levels, the operation level which compares the same operations of the SUT versions, and the global level which compares one overall measurement of all SUT versions. This overall measurement is based on all operations from an SUT version.

6.1.2. Experiment Settings

The section discusses the experiment settings of the evaluation. First of all, the hardware and software specifications of the used systems in Section 6.1.2.1. In the following sections, the used applications with the corresponding scenarios are explained. Section 6.1.2.2 describes the Broadleaf Heat Clinic[1] and Section 6.1.2.3 covers the test application which provides synthetic measurements.

[1]Broadleaf Heat Clinic: https://github.com/BroadleafCommerce/DemoSite

Table 6.1.: Hardware and software specification from the VM which is used in the evaluation.

Operating system	Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-134-generic x86_x64)
RAM	8 GB
CPU cores	4
CPU GHz	2.3
Disk space	78 GB
Java	OpenJDK 9
Docker	18.06.1-ce
Maven	3.6.0

6.1.2.1. Hardware and Software Specification

The evaluation is triggered by a system with manages the model transformations, the Docker deployment, the execution of the load tests with JMeter, the regression analysis, and the report generation. The SUT is deployed on a virtual machine (VM) which has the specifications described in Table 6.1. JMeter from the evaluation executor performs the load tests on the VM.

6.1.2.2. Broadleaf Heat Clinic

For the evaluation, the Spring Boot application Broadleaf Heat Clinic is used which is available on GitHub. It consists of a set of microservices which each contains an own REST API. For the evaluation, we use modified versions of this application.

For each execution of the Heat Clinic, we use the same load test configurations. For each SUT version, the configurations are the following:

- 15 minutes as load test duration,
- 200 users, and
- 50 seconds as ramp up time.

The load test has a duration of 15 minutes because the response time of the operations is stabilized over the time to reduce the variability of the measurements. The 200 users mainly need between 25 % and 40 % CPU usage of the VM and need 75 % CPU usage during peaks. Fewer users need less CPU usage, which decreases the probability that effects can be observed. More users have the effect that the CPU usage of the peaks increases which results in unpredictable side effects. To reduce the CPU usage during

the start-up period, we use 50 seconds as ramp up time which means that every second 4 new users are added.

To use the Heat Clinic versions, an appropriate ContinuITy model is necessary for each version. Since the Heat Clinic is already used with ContinuITy, the IDPA models and a random generated Markov chain are existing for each version. Appropriate WESSBAS models are not existing for the versions, but one of the first steps of the approach is to transform a provided WESSBAS model into a behavior model. This step can be omitted. We use the random generated Markov chain instead of the WESSBAS model. This Markov chain is in CSV format and can be transformed into an appropriate behavior model.

For each version, the behavior model and both IDPA models are uploaded to ContinuITy via REST interfaces. A BenchFlow DSL template is used to retrieve the fulfilled BenchFlow DSL from the *BenchFlow* service of ContinuITy. The template contains the SUT versions and the regression goal which should be evaluated.

For each scenario of this evaluation, we conduct two regression tests. The properties of a regression condition is explained in Section 4.2.2.2. The first regression test is on the operation level with the following regression conditions:

- the metric is median response time,
- the absolute delta is 50 ms, and
- the percent delta is 25 %.

The second regression test is on the global level with the following regression conditions:

- the metric is median response time,
- the absolute delta is 25 ms, and
- the percent delta is 10 %.

For the evaluation with the Heat Clinic, we use three different scenarios. The scenarios cover different functionalities of the approach. The approach supports two different regression methods. The individual method is covered by the first scenario, and the intersection method is covered by the second scenario. A representative user behavior is not existing for a new SUT version. The third scenario covers this case using the intersection method.

The scenarios are the following:

- Scenario 1: Individual regression load test

  For this scenario, the application Heat Clinic is used with ten different versions. These versions are used for load testing with the individual regression as goal type.

The oldest version is 2, and the newest version is 20. To monitor the measurements of the versions, it does not matter how to combine the SUT versions in a regression test because each version always uses the corresponding behavior. The benefit is that we only have to run the load test once for each version and after that, we can individually combine the load test results for the regression tests. We combine all ten versions into one regression test because this regression test contains all regressions which can be detected if we use a subset of versions.

We expect to find no regressions on the operation level because the changes of the versions are mainly API changes, that means added and removed REST methods. Also, minor changes inside the REST methods do not have a high performance impact. On the global level, we expect to find two regressions between the oldest versions and the newest versions because in the newest versions are more methods with higher complexity.

- Scenario 2: Intersection regression load test

  In this scenario, we use only a subset of versions because each combination of the versions requires new load tests, and the added value for more combinations is low. The regression condition can be changed after each regression test without executing the load tests again. but we can change the regression condition afterwards for each regression test. In total, we run three intersection regression tests. Two times with two versions and one time with three versions. The used versions are in the first run the oldest version and the newest versions, means version 2 and 20. The second run contains version 5 and 10, and the last run contains three versions which are the three newest versions 18, 19, and 20.

  We expect to find no regressions on the operation and global level.

- Scenario 3: Intersection regression load test with partly missing behavior

  As in scenario 2, we can only change the regression condition after executing the load tests. To allow a comparison between the approaches with and without behavior, we use the same intersection regression tests as in scenario one but without the behavior of the newest version. Therefore we have two times two versions with one behavior each and one time three versions with two behaviors.

  We expect to find no regressions on the operation and global level.

6.1.2.3. Test application

To evaluate the regression analysis in more detail, a self-made test application is used. This test application provides synthetic measurements with self-defined performance

usages on operation level. Table 6.2 shows the used SUT versions with the corresponding operations. If an operation is available in an SUT version, it has either an *L* for low performance usage or an *H* for high performance usage. All operations do the same but they contain different sleep durations. Low performance usage means that an operation contains a sleep duration between 20 ms and 30 ms. Operations with a high performance usage contains a sleep duration between 200 ms. and 210 ms. The response time of an operation is the sleep duration and the normal execution time.

Four SUT versions are used with five operations in total. For example, version 3 consists of two operations, one operation with high and one operation with low performance usage. All operations contain transitions to all operations with the same probabilities and think times.

For each test run, a self-written BenchFlow DSL is used to execute the BenchFlow workflow directly. The following configurations are used for the load tests:

- 2 minutes as load test duration,
- 50 users, and
- 30 seconds as ramp up time.

As in the Heat Clinic from Section 6.1.2.2, two regression tests are conducted.

For the operation level, the following regression conditions are used:

- the metric is median response time,
- the absolute delta is 50 ms, and
- the percent delta is 25 %.

The second regression test with the global level uses the following regression conditions:

- the metric is median response time,
- the absolute delta is 25 ms, and
- the percent delta is 10 %.

For the evaluation with the test application, the same scenarios are used as for the Broadleaf Heat Clinic in Section 6.1.2.2.

- Scenario 4: Individual regression load test

  In this scenario, all four versions of the test application are compared with the individual method.

  We expect to find two regressions on the operation level. The operation *C* from version 3 should lead to both regressions. We expect to find four regressions on the

Table 6.2.: Experiment setup of the test application. *L* means that the operation needs low performance usage in particular SUT version, *H* means high performance usage, and - means that the operation is not existing in the particular version.

	Operations				
SUT version	A	B	C	D	E
1	L	L	L	-	-
2	L	-	L	H	-
3	L	-	H	-	-
4	L	-	-	L	H

global level. The version 2 should be responsible for one regression, the version 3 for two regressions, and the version 4 for one regression with the version 1.

- Scenario 5: Intersection regression load test

  Each version is compared with all older versions but in different regression tests and not in one test. For this scenario, the intersection method is used.

  We expect to find two regressions on the operation level. The operation *C* from version 3 should lead to both regressions. We expect to find two regressions on the global level. The version 3 should be responsible for both regressions.

- Scenario 6: Intersection regression load test with partly missing behavior

  This scenario is divided into two runs. In the first run, the SUT version 3 is used without behavior, and in the second run, the SUT version 4 is used without behavior. In both runs, the older versions are compared with these versions in a regression test with the intersection method.

  We expect to find two regressions on the operation level. The operation *C* from version 3 should lead to both regressions. We expect to find two regressions on the global level. The version 3 should be responsible for both regressions.

6.1.3. Description of Results

This section describes the results of each scenario.

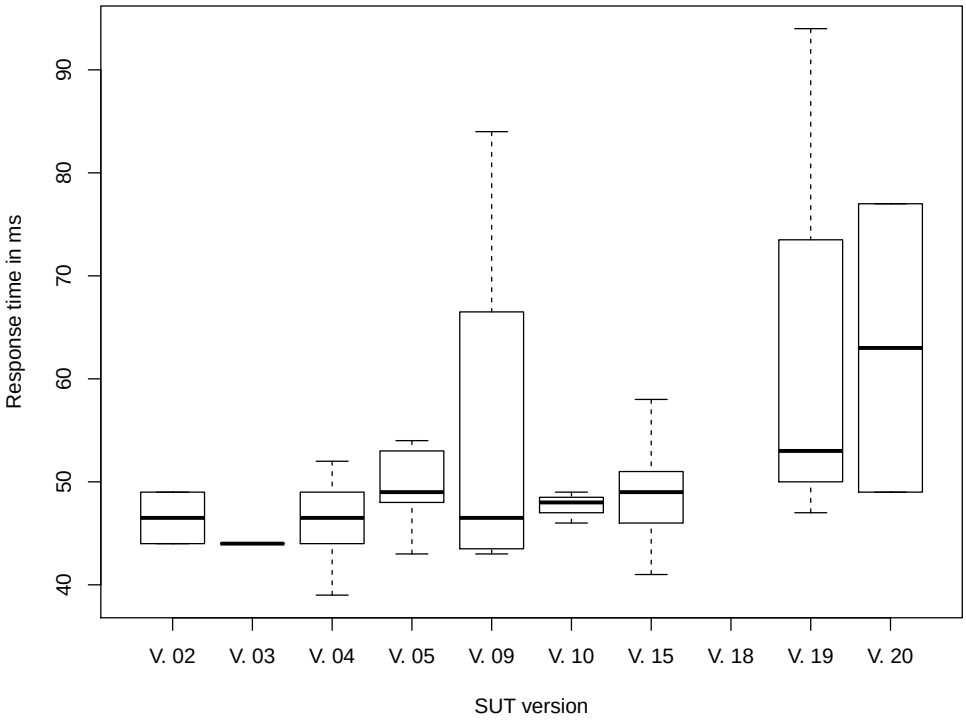


Figure 6.1.: Results of scenario 1: Measurements of a single operation.

6.1.3.1. Scenario 1

In the first scenario, we compare ten versions of the Heat Clinic with 54 operations in total. 51 of them contain measurements of which 44 operations contain at least measurements of two versions. The first regression test on the operation level detected 15 regressions in five operations. Figure 6.1 visualizes the measurements of an operation which contains seven regressions. In all cases, the newest version is responsible for the regressions. The variability of the measurements are due to too few measurements. In addition to that, version 18 contains no measurements for this operation although the operation is a part of the version.

Another operation contains five regressions, whereby the version 9 is responsible for four regressions which contain all older version from version 9. A further operation contains one regression which is based on two measurements, one for each version.

The second regression test on the global level detected two regressions between the oldest version 2 and the two newest versions 19 and 20. Figure 6.2 shows box plots each with the measurements of a version. The red dashed lines define the mean of the measurements. In this figure, an increase of the response time is visible for the mentioned versions that lead to regressions.

Table 6.3.: Results of scenario 1.

Level	No. of detected regressions	No. of existing regressions
Operation	15	0
Global	2	2

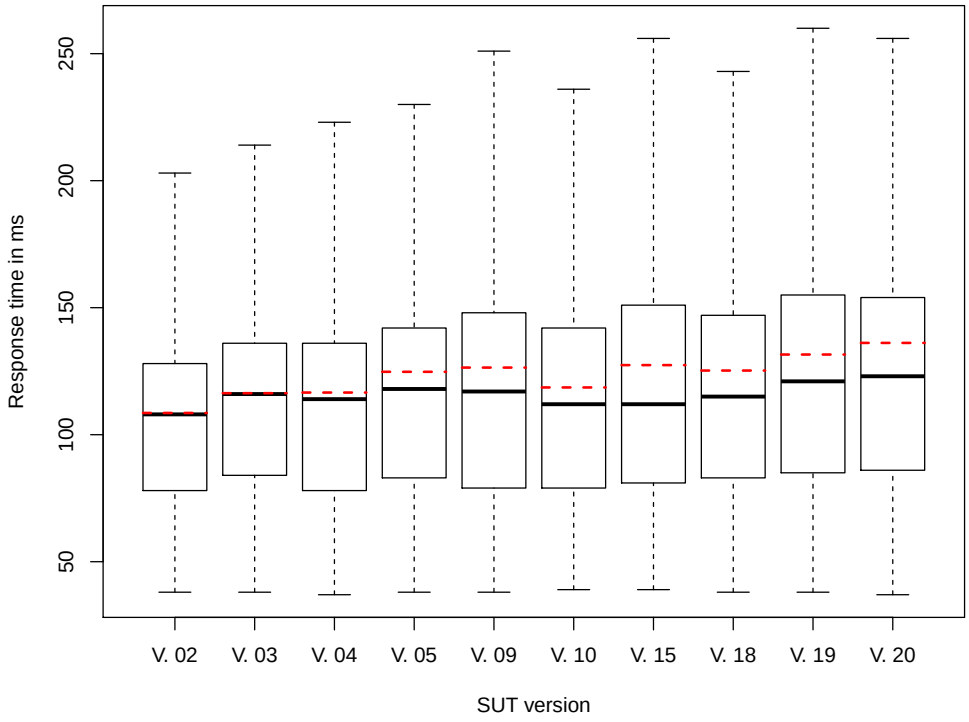


Figure 6.2.: Results of scenario 1 on the global level.

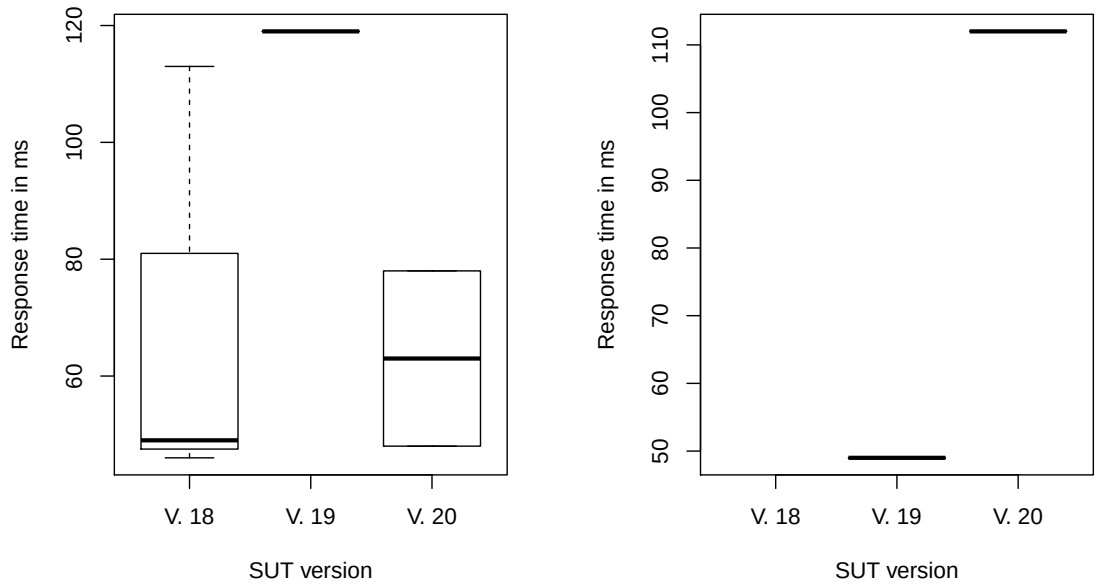
6.1.3.2. Scenario 2

The second scenario is divided into three runs. We compare two versions on the first two runs, and three versions in the third run. Each run uses the intersection method.

Each run detects no regression for both regression tests. The first run with version 2 and 20 contains 20 operations with at least one measurement for each version. In total, 24 operations are available in the intersection. The second run with version 5 and 10 has 26 operations with at least one measurements for each version, whereby 32 operations are available. In the third and last run, we have 45 operations in total, 32 of them contain at least one measurement for each version.

Table 6.4.: Results of scenario 2.

Level	No. of detected regressions	No. of existing regressions
Operation	0	0
Global	0	0



(a) Results of scenario 3 with two detected regressions.

(b) Results of scenario 3 with one detected regression.

Figure 6.3.: Results of scenario 3 on the operation level.

6.1.3.3. Scenario 3

In the third scenario, regression tests are executed with missing behaviors for the newest version.

The first two runs, each consisting of two versions, did not detect regressions for both regression tests. The third run detected three regressions on the operation level. The regressions are detected in two operations which are shown in Figure 6.3. In Figure 6.3a, two regressions are existing. The regressions are between the versions 18 and 19, and between 18 and 20. Version 19 contains only one measurement, and version 20 contains two measurements. The third regression is shown in Figure 6.3b. Both involved versions contain only one measurement for this operation. The variability of the two measurements are due to too few measurements. The third version does not contain

measurements for this operation although the operation is a part of the version. The regression test on the global level did not any detect regressions.

Table 6.5.: Results of scenario 3.

Level	No. of detected regressions	No. of existing regressions
Operation	3	0
Global	0	0

6.1.3.4. Scenario 4

This scenario compares all versions with the individual regression method.

The first regression test on the operation level detected two regressions on operation *C*. Figure 6.4a shows a diagram with box plots for this operation wherein the regressions can be detected.

The second regression test on the global level detected two regressions. In Figure 6.4b, the overview is visualized including the mean of each box plot. The regressions are detected between the versions 1 and 3, and between 2 and 3.

Table 6.6.: Results of scenario 4.

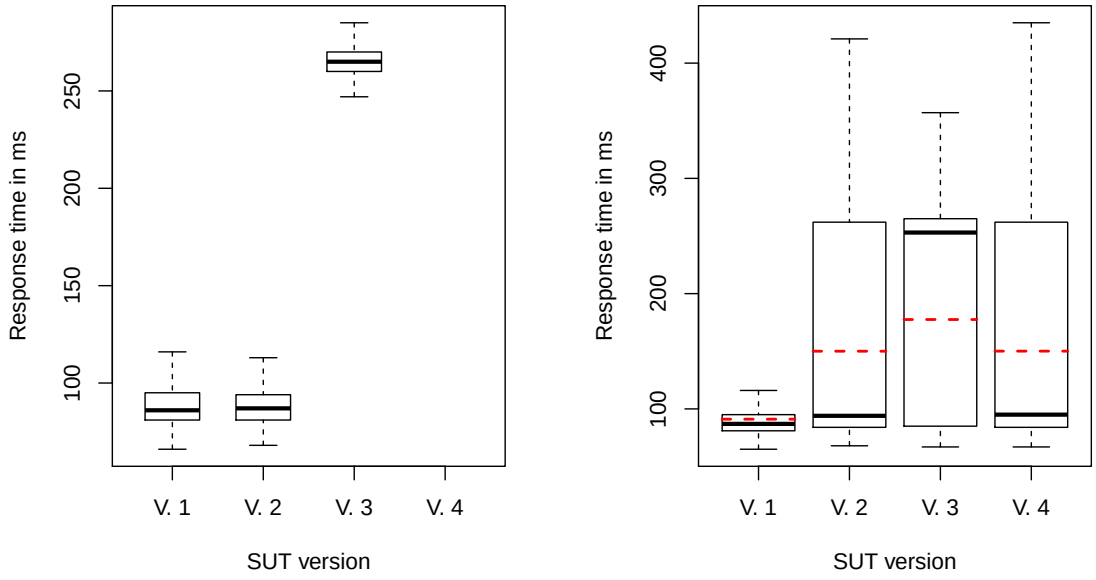
Level	No. of detected regressions	No. of existing regressions
Operation	2	2
Global	2	4

6.1.3.5. Scenario 5

This scenario compares the versions of the test application with the intersection method. The results of the six regression tests for each level are depicted in Table 6.8. The regression tests on the operation level detected two regressions for version 1 and 3, and for version 2 and 3.

Also, the regression tests on the global level detected two regressions for the same versions as in the regression test on the operation level. Figure 6.5a visualizes the measurements of the versions 1 and 3.

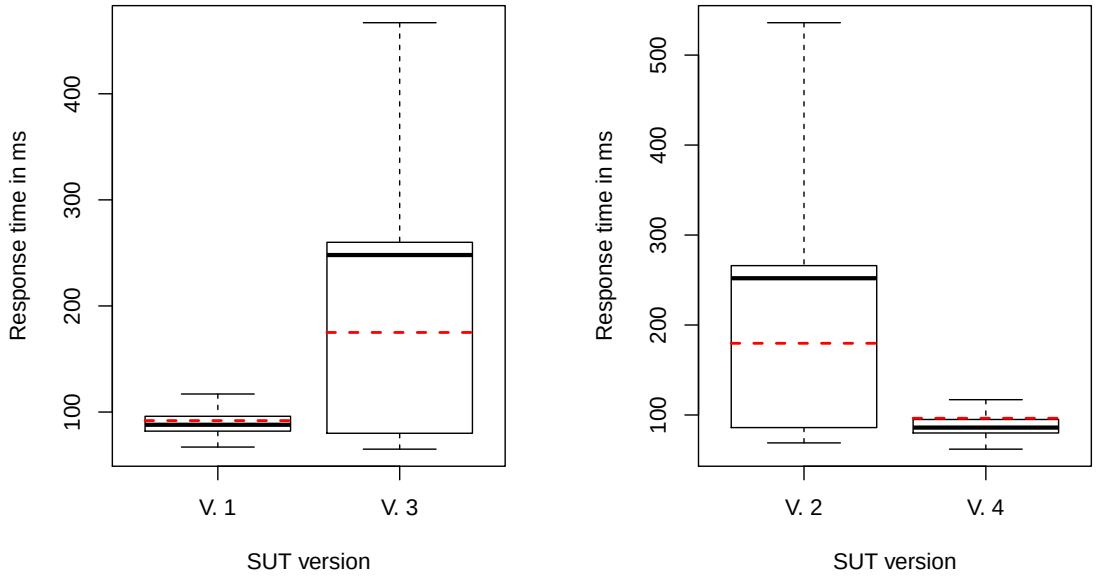
The comparison between the versions 2 and 4 shows performance improvements on the operation and on the global level. Figure 6.5b shows the improvements on the global level.



(a) Results of scenario 4 on the operation level with a single operation.

(b) Results of scenario 4 on the global level.

Figure 6.4.: Results of scenario 4.



(a) Results of scenario 5 with version 1 and 3 on the global level.

(b) Results of scenario 5 with version 2 and 4 on the global level.

Figure 6.5.: Results of scenario 5 with the intersection method using the global level.

Table 6.7.: Results of scenario 5.

Level	No. of detected regressions	No. of existing regressions
Operation	2	2
Global	2	2

Table 6.8.: Detected regressions of scenario 5. *X* means that a regression was detected and - means that no regression was detected for the version on the left compared to the version on the top of the table.

	SUT version		
SUT version	2	3	4
1	-	X	-
2		X	-
3			-

6.1.3.6. Scenario 6

The scenario is divided into two runs. In the first run, the SUT version 3 without corresponding behavior is used. In the first regression test on the operation level, one regression is detected for each older version that means in total two regressions. Also, the second regression test on the global level detected one regression for each older version, two regressions in total.

In the second run, the version 4 is used without corresponding behavior. For both regression tests, no regressions are detected instead of this, the comparison between version 2 and version 4 shows performance improvements on the operation and the global level.

Table 6.9.: Results of scenario 6.

Level	No. of detected regressions	No. of existing regressions
Operation	2	2
Global	2	2

6.1.4. Discussion of Results

This section discusses the results of each scenario.

6.1.4.1. Scenario 1

The first scenario with the Heat Clinic detected 15 regressions on the operation level. We did not expect to find any regressions in the first scenario. Therefore, all 15 regressions are false positive results. To find the root cause of the regressions, we look into the measurement results. The newest version, which is responsible for the regressions, has only two measurements in 12 of 15 regressions. In the other three regressions, the responsible version has only one measurement. There are also missing measurements for operations which are available in a version. Therefore, the version is handled like it does not contain the certain operation, and hence, the version is not used for the regression analysis. If no measurements are existing for an operation in a newer version, this can lead to false negative results, because the operation from the older version can perform better than from the newer version. Such a case was not found in this scenario.

The reason for the few measurements of some operations is that in some versions, a call to this operation is unlikely because the probability to call this operation is low. A higher duration of the load tests can reduce the problem but cannot solve it because the probabilities are the same, only the possibilities increases by the increased duration. For example, an operation of an SUT version contains one measurement in the duration of 10 minutes. If the duration is doubled to 20 minutes, this does not lead to two measurements automatically because the call of an operation is based on probabilities. Therefore, zero or more measurements can occur. A further increase of the duration leads to even more possibilities. Neither two nor four measurements are representative for an operation. Therefore, these results are not really usable.

The second regression test on the global level detected two regressions which match to our expectations. The reason for the regressions is that new operations are used which have higher performance usage as the other existing operations in the other versions.

6.1.4.2. Scenario 2

The regression results of this scenario are as expected. As discussed in Section 6.1.4.1, the probabilities of the operations can result in false results. For example, in the third run with three versions are measurements for 38 operations available, but 45 operations are existing for the intersection method. In addition to that, four operations contain measurements of only one version, and two operations contain measurements of two versions, whereby all operations from the intersection method should contain measurements for all versions.

6.1.4.3. Scenario 3

The last scenario with the Heat Clinic was conducted with the intersection method and a missing behavior of a tested version. The results are not as expected because scenario 3 detected three regressions on the operation level, these are false positive results. Responsible for these regressions are versions with only one measurement for the operation. As discussed in Section 6.1.4.1, the reason for the results is the probabilities of the operations.

6.1.4.4. Scenario 4

The results of the first regression test on the operation level are as expected. The operation with the high performance usage is correctly detected.

In the second regression test on the global level, the results are not as expected. We expected that two more regressions are detected. In two versions, operations with high performance usage are used, whereby these operations have no effect on the detected regressions. The reason is the distribution of the measured values. The first version contains three operations with low performance usage and therefore the measured values consist exclusively of low usage. In the two versions, which we expected are responsible for a regression, there are two operations with low and one operation with high performance usage. Thus about 66 % of the data correspond to a low usage. A median produces a low value because more than 50 % of the measured values have a low usage. Therefore, if the median is used as the metric in this scenario, some regressions may not be detected. These regressions are false negative results. If the average is used as the metric instead of the median, the regressions are detected, but the average can also detect false positive regressions caused by outliers.

6.1.4.5. Scenario 5

The regression analysis detected two regressions on the operation level, and two regressions on the global level detected. These detected regressions are as expected. The reason for the regressions is that version 3 contains the operation *C*, as well as version 1 and 2. The difference is that the operation *C* in version 3 has a high performance usage. The intersection contains this operation, and the regression test on operation level detected two regressions, each for version 1 and 2. This operation is also the cause for the regressions on the global level.

We determined performance improvements from version 2 to version 4 on the operation level and on the global level. The reason for the improvements is that the versions 2 and

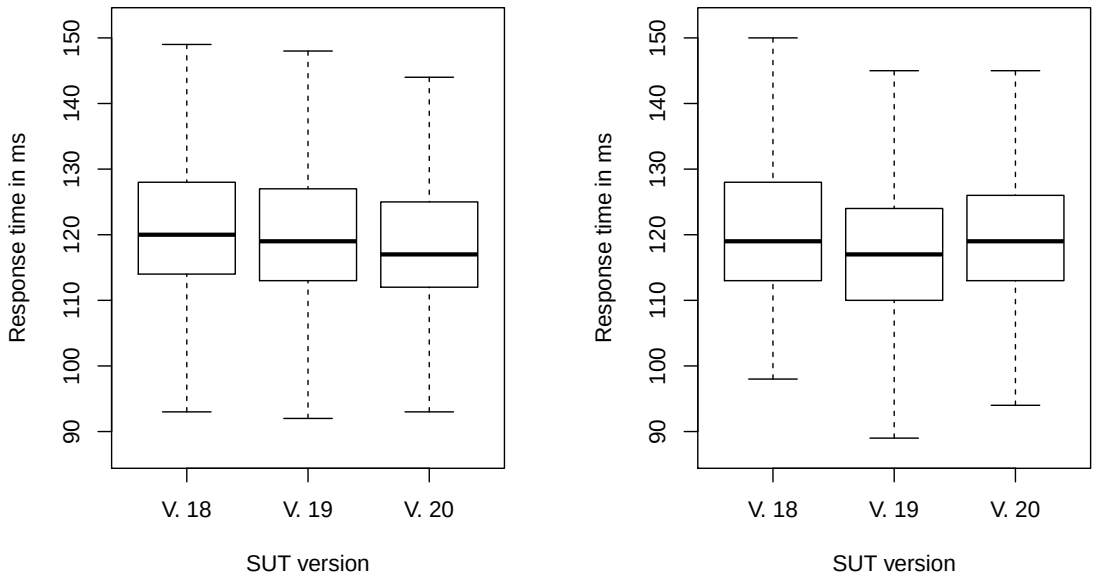
4 contain the same operation *D*, which has a low performance usage in version 4 and a high performance usage in version 2. On the global level, the other high performance usage operation from version 4 is not considered because it is not available in version 2. That leads to that the performance seeming improves on the global level. Compared to scenario 4 which compares both versions with the corresponding behavior, a constant performance usage is visible on the global level. Figure 6.4b shows the measurements of scenario 4 using the individual method, and Figure 6.5b visualizes the measurements of scenario 5 using the intersection method. Therefore, the improvement of the global level can lead to wrong expectations for version 4.

6.1.4.6. Scenario 6

The last scenario was divided into two runs, and both runs used the test application. In the first run, the SUT version 3 is used without a corresponding behavior, and in the second run, the version 4 is used without a corresponding behavior. The results of scenario 6 are the same as for scenario 5. The reason is that the same operations are used, and the behavior is not changed because all versions from the test application use the same behavior.

6.1.5. Conclusion

Before we answer RQ1, the underlying questions of RQ1 are answered. The first research question is RQ1.1. The scenarios 1 and 3 show that the regression analysis detects false regressions. These regressions are false positive results. The reason for all of these regressions is that the probabilities of some operations are too low. This leads to no or few measurements for an operation in a specific SUT version. A longer duration of the load test can reduce the problem but cannot solve the problem because a longer duration does not guarantee more measurements. The same cause can lead to undetected regressions which are false negative results. The reason is that measurements are not available for all operations because the operation was not executed in the corresponding version. However, the version contains the operation. The scenarios 1 and 2 show that measurements are missing for operations. In the scenarios 4, 5, and 6, measurements are available for all operations. In these scenarios, the regression analysis correctly detected all regression except for two. The two regressions are false negative results because they were not detected. If the condition metric is changed, these regressions can be detected. The number of false negative and false positive regressions depend on the selected regression conditions. The analysis detected false negative and false positive regressions, but the number can be reduced by the mentioned changes.



(a) Results of scenario 1 with the individual method on the operation level.

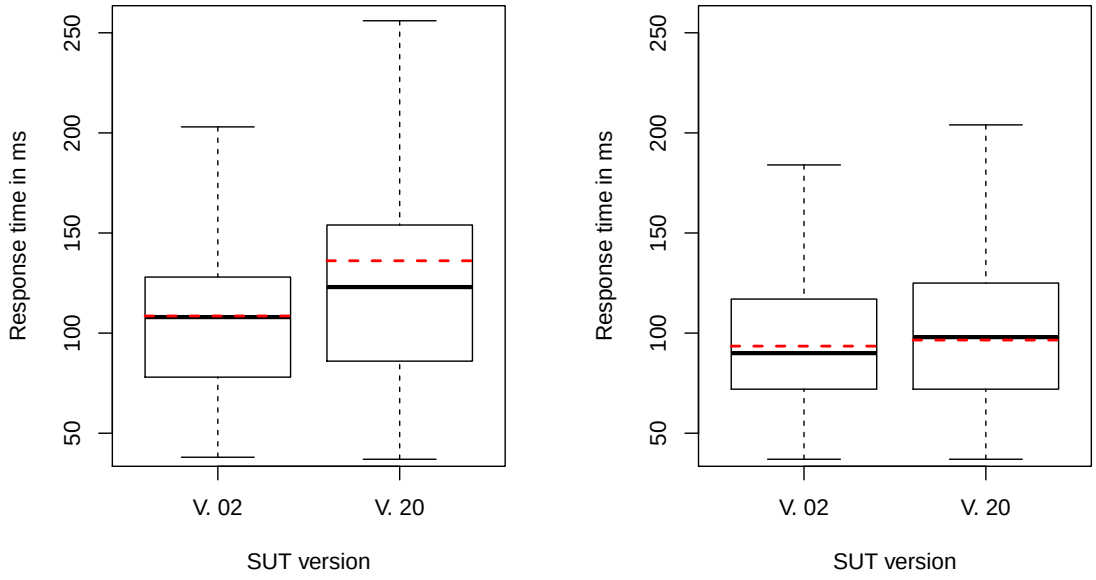
(b) Results of scenario 2 with the intersection method on the operation level.

Figure 6.6.: Comparison of the individual and intersection method on the operation level.

Scenarios 3 and 6 cover the RQ1.2. As discussed in RQ1.1, three false positive regressions are detected which are caused by the low amount of measurements. The other regressions are correctly detected. This includes regressions on the operation level and on the global level. Therefore, the regression analysis detects regressions on versions without existing representative user behaviors.

To sum up and to answer RQ1, the regression analysis helps to find regressions. Some regressions are only visible on the global level and some on the operation level. The regression analysis should cover regression tests for both levels. In addition to that, several metrics should be used as showed in scenario 4. Therefore, in order to detect all regressions, it is better to obtain more regression tests with different conditions to avoid false negative results representing missing regressions, and to accept potentially false positive results. An undetected regression can lead to a performance degradation that is not detected. Nevertheless, the scenarios show that the most regressions are correctly detected as already discussed in RQ1.1, and that the analysis detected regressions in SUT versions without representative user behavior as already discussed in RQ1.2.

For RQ2, the measurements of the individual and the intersection method are compared. Therefore, the first two scenarios of both applications are used that means scenario 1, 2, 4, and 5. On the operation level, the measurements are similar because a behavior



(a) Results of scenario 1 with the individual method on the global level.

(b) Results of scenario 2 with the intersection method on the global level.

Figure 6.7.: Comparison of the individual and intersection method on the global level.

does not change the performance usage of an operation, as shown in Figure 6.6. All measurements of a version form the global level. This global level can be different in the individual and intersection method. For example, the versions 2 and 20 are used in the first scenario with the individual method, and in the second scenario with the intersection method. Figure 6.7 visualizes the measurements of both scenarios using the versions 2 and 20. In the individual method, the measurements of both versions are different, whereas the measurements in the intersection method from both versions are similar because the intersection leads to that the same operations are in the versions 2 and 20. Therefore, the measured results on the operation level are similar, whereas the measured results on the global level differ.

For RQ2.1, we compare the detected regressions of both methods. The difference of the measured results on the global level leads to a regression in the individual method but not in the intersection method. This is not the case for the versions 5 and 10 of the same scenarios because both scenarios did not detect regressions for these versions. Nevertheless, the scenarios show that different regressions can be detected with different regression methods but only on the global level and not on the operation level.

6.1.6. Threats to Validity

This section describes the different threats to validity of this research. Internal, external, construct, and conclusion validities are considered.

Conclusion validity: The number of users can be too low or too high to represent real load. More users lead to more measurements and more significant measurements because the VM is more under load and therefore needs more CPU usage. Too many users lead to too high CPU usage which results in unexpected side effects. Before the execution of the scenarios of the Heat Clinic, we determined the CPU usage from the VM with the Heat Clinic and different numbers of users to find the optimal number of users. The used 200 users need up to 40 % CPU usage in average and 75 % during peaks.

The duration of the load tests can be too low to represent real load tests. A longer duration leads to more significant measurements because the SUT is better represented. We used a duration of the load test in which the response time of the operations stabilized over the time.

The measurements of the SUT versions contain no or too few measurements for some operations. These regression tests lead to wrong expectations and wrong conclusions. The cause of this problem is that the probabilities for some operations are too low. A longer duration of the load tests can be used to reduce this problem, but this is not a guarantee for more measurements. A second possible solution is an adaption of the behavior to increase the probabilities of the operations with too few measurements. The change of the representative user behavior leads to a more precise regression analysis. This solution includes a consideration of the load test duration, the change of the representative user behavior, and the possibility to detect regressions. For this thesis, the solution is out of scope and therefore a part of the future work.

Internal validity: The results of the SUT applications were known before the execution of the approach, this can lead to other effects being less considered and might remain unnoticed. For the evaluation of the results, the detected regressions and also the aggregated measurements are used to prove the detected regressions and to prove if other effects occur which were not previously known.

A distributed experiment setup is used with the SUT on a VM and the executor of the load tests on another system. This distributed experiment can cause network delays which lead to wrong measurements and therefore in wrong regression detections. Both systems were in the same network with high Internet speed. Also, both systems have only executed the processes which are necessary for the evaluation to reduce side effects by other processes.

Construct validity: In the self-made test application, the operations with the high performance usage were previously known. This leads to bias of the results during the experiment with the test application. With the Broadleaf Heat Clinic, we know that the operations are only changed a little bit which results in a bias of the results. All results of the evaluation are checked despite the bias.

For all regression tests, the same regression conditions are used but maybe are different regression conditions with different metrics better to detect regressions. The regression conditions were selected according to the detected regressions on a regression test of the Heat Clinic before the evaluation. The conditions are chosen so this the regression test detected all regressions correctly.

External validity: The results of the experiment cannot be generalized because of the limited number of used applications. We selected a representative benchmark, namely the Heat Clinic, to cover a large field of applications. In addition to that, the self-made test application is used, which provide synthetic measurements, to prove the regression analysis under different conditions.

6.2. Evaluation of the BenchFlow DSL

This section evaluates the changes of the BenchFlow DSL. First of all, the research questions are explained in Section 6.2.1. After that, the methodology is envisioned in Section 6.2.2 for achieving the evaluation goals. The results of the comparison are described in Section 6.2.3 and in Section 6.2.4, the results are discussed and the research questions are answered.

6.2.1. Research Questions

This section describes two research questions (RQ) of the BenchFlow DSL.

RQ3: What are the advantages and disadvantages of the implemented extensions compared to related approaches?

This question evaluates the BenchFlow DSL compared to the ContinuITy model and the JMeter model. The ContinuITy and the JMeter model are used in this approach. This question deals with similarities and differences between these three models. The main focus of the comparison are the extensions of the BenchFlow DSL which are the HTTP workload, the regression part, and the deployment properties.

RQ4: How can the implemented extensions be improved?

The second question aims to allow the usage of the BenchFlow DSL extensions in more use cases and fields of application. The question deals with improvements of the BenchFlow DSL extensions which are made during this thesis. Extensions are the HTTP workload, the regression part, and the deployment properties.

6.2.2. Methodology

To answer the research questions, features are considered which are added or not added by the extensions of the BenchFlow DSL. In addition to the features, the effort to modify the HTTP workload is measured by the Lines of Code (LOC). We compare the BenchFlow, the ContinuITy, and the JMeter model but not the tools themselves. In the next step, the models are compared with questions about the considered features. For the RQ3, the advantages and disadvantages of the BenchFlow model are derived from the comparison results. The RQ4 is answered based on the missing features or rather on the disadvantages of the BenchFlow model. The answer to RQ4 contains possible improvements for the missing features.

We consider eight features and four modifications of the HTTP workload. The features cover the three extensions. The first four questions cover the HTTP workload extension, consisting of the definitions from (1) the IP address, (2) the behavior, and (3) the extraction. The next two features are about the regression extension, consisting of the definitions from (1) the static regression and (2) the version-specific regression. One further features covers the deployment properties extension. The last two features are about the model files, consisting of (1) the number of files and (2) the data formats of the files.

To measure the effort, the four modifications of the HTTP workload are selected so that the main functionalities of the HTTP workload are covered. The main functionalities are (1) changing the initial state, (2) add a new operation, (3) change the behavior of a single operation, and (4) add a new parameter. We omit the removing functionalities because they are similar to the adding functionalities. To compare the effort based on the LOC, we use well-formatted models.

In the following list, eight questions about features, and four questions about the effort to modify the HTTP workload are listed. These questions are used for the comparison of the three models. Each feature can be identified by an abbreviation for the question. In the following section, these abbreviations are used.

1. **IP:** Is the definition of different IP addresses supported?
2. **Behaviors:** Are different behaviors for the same set of operations supported?

3. **Extraction:** Are regular expression and JSON extractions from a response supported?
4. **Static regression:** Is the definition of static regression conditions supported?
5. **Version regression:** Is the definition of regression conditions supported to compare different version?
6. **Deployment:** Is the definition of deployment properties of the SUT supported?
7. **Files:** How many files are at least necessary for a test with a behavior?
8. **Data format:** Which data formats are used?
9. **Change initial state:** How much effort is it to change the initial state?
10. **New operation:** How much effort is it to add an HTTP operation without corresponding behavior?
11. **Change behavior:** How much effort is it to change the behavior, including the think times, of a single HTTP operation?
12. **New parameter:** How much effort is it to add a new query parameter with one value to an HTTP operation?

6.2.3. Description of Results

In the comparison, the native ContinuITy model is used which consists of the IDPA annotation, IDPA application, and the WESSBAS model. The ContinuITy model which is used in this research consists of the IDPA models and the behavior model which is a subset of the WESSBAS model.

Table 6.10 shows an overview of all answers of the feature questions and Table 6.11 depicts the results of the effort comparison.

1. **IP:** The definition of the IP address of the SUT is defined in the BenchFlow DSL on a global level for all operations, thereby only one service with an IP address can be load tested. ContinuITy defines the IP address on each operation, whereby different services with different addresses can be used. JMeter provides the possibility to define the addresses on a global level or on the operation level. Therefore, different addresses can be used.

Table 6.10.: Feature comparison of the BenchFlow, the ContinuITy, and the JMeter model. The "*" in the column JMeter means that this answer is affected by the JMeter extension *Markov4JMeter* which is used in this approach.

	Models		
Questions	BenchFlow	ContinuITy	JMeter
1 - IP	✗	✓	✓
2 - Behaviors	✓	✓	✓*
3 - Extraction	✓	✓	✓
4 - Static regression	✗	✗	✓
5 - Version regression	✓	✗	✗
6 - Deployment	✓	✗	✗
7 - Files	1	3	2*
8 - Data formats	YAML	YAML, XML	XML, CSV*

Table 6.11.: Effort comparison of the BenchFlow, the ContinuITy, and the JMeter model. The effort is measured by changed LOC. *N* means the number of HTTP operations, and *M* means the LOC of an operation.

	Models		
Questions	BenchFlow	ContinuITy	JMeter
9 - Change initial state	2, (N+2M)	1	2
10 - New operation	(N+4)	25	(N+95)
11 - Change behavior	1	2-3N	1
12 - New parameter	1, 2	5-9	4, 6

2. **Behaviors:** BenchFlow supports only the definition of one user behavior for a set of operations. Adding a second behavior to the already existing operations is not possible. To achieve this, the operations have to be copied to form a new set of operations. ContinuITy defines the behaviors in the WESSBAS model, which has the benefit to define multiple behaviors for the same operations. JMeter does not natively support behaviors but in this research, an extension is used. This extension supports different behaviors for the same set of operations.

3. **Extraction:** All three models support the extraction with regular expressions and JSON expressions. JMeter supports to define the field of the extraction, e.g., body or headers. BenchFlow and ContinuITy support only the extraction of the body.

4. **Static regression:** BenchFlow supports to define regression conditions based on the results of previous SUT versions but not static regressions such as that the

response time has to be lower can 1 second. ContinuITy does not support to define regression conditions. JMeter supports to define regressions with assertions containing specified values, e.g., duration assertion with response time and response assertion with response headers [ASP18].

5. **Version regression:** BenchFlow supports the definition of regression conditions to compare different SUT versions. For example, regression detection is allowed on the operation level and global level. ContinuITy does not support to define regression conditions. JMeter does not support the definition of regression conditions to compare different versions.

6. **Deployment:** The BenchFlow DSL supports to store information about the deployment of the SUT. In the models from ContinuITy and JMeter, the definition of deployment information is not possible.

7. **Files:** The HTTP workload, the regression part, the deployment properties, and the rest of the BenchFlow DSL are described in a single YAML file. BenchFlow requires no other files. The YAML file can contain HTTP workloads and behaviors for different SUT versions which are both required for regression testing. The native ContinuITy model consists of the IDPA annotation, the IDPA application, and the WESSBAS model which are all stored in a separate file. Each version needs these three files, means if three versions are used up to 9 files are required because the files can be reused if a particular file matches to another version. JMeter has a test plan and for each behavior, a behavior model file.

8. **Data format:** As described in the previous question, the BenchFlow DSL is one YAML file. ContinuITy has three models consisting of the IDPA models as YAML files and the WESSBAS model as an XML file. The test plan of JMeter is an XML file, and the behavior model files are CSV files.

9. **Change initial state:** The initial state in BenchFlow cannot be changed without changing the behavior. BenchFlow always uses the first operation in the list of operations. Changing the order of operations leads to that the Markov chain does not match to the order of the operations. Therefore, a new initial state requires changes in the behavior. In the following, N defines the number of operations and M the length of the new operation. To exchange the first operation, twice as much LOC have to be changed than the LOC of the new initial operation. The reason is that the new initial operation has to be deleted and inserted. For this $2M$ LOC changes are needed. In the behavior, the positions of the transitions change according to the new order of the operations. This requires N LOC changes. In total, $N+2M$ LOC changes are necessary.

   Another solution is that the HTTP workload does not contain a real HTTP operation as the initial state. Instead, an HTTP operation placeholder which is not handled as

a normal HTTP operation but it contains a behavior. The benefit of this placeholder is that different operations can be used as start states. The start state is selected by the probability of the placeholder behavior. In this case, a change of the initial state requires to adjust the behavior of the placeholder. The effort is to remove the probability from the transition that targets the old initial state, and to add the probability to the transition that targets the new initial state. In total, two LOC has to be changed.

In the ContinuITy model, the initial state is a part of the WESSBAS model. A single field defines the initial state with an operation name. To change the initial state, the value of the field has to be changed to another operation name. The change of an initial state requires an adaption of one LOC.

The initial state of the JMeter model is defined in the separate CSV file with a "*" in the row header. This symbol can be added to another operation, which leads to a new initial state. The change of an initial state requires an adaption of two LOC.

10. **New operation:** In the following, *N* defines the number of operations.

    The BenchFlow model requires for each new operation at least four properties, namely (1) the operation ID, (2) the endpoint, (3) the HTTP method, and (4) the protocol. For the properties, four LOC are required. Each new operation requires a set of transitions which targets all existing operations. Also, the sets of the other operations have to be adjusted because each operation has to contain transitions to all operations. For the set of the new operation, one LOC change is required because all transitions from this operation are in one line. For each set from the other operations, a transition to the new operation is necessary. This leads to one additional LOC for each other operation. In total, a new operation requires *N+4* LOC changes.

    A new operation requires changes in all three ContinuITy models. In the IDPA application model, the operations are defined. For the properties of the operations, at least six LOC are necessary including the reference to the IDPA annotation model. In the annotation model, three additional LOC are necessary. The definition of the operation in the WESSBAS model requires 16 LOC. In total, 25 LOC are required to define a new operation.

    The JMeter model needs a new Markov state before a new operation can be added. This state leads to 75 LOC changes, and the new operation requires another 20 LOC changes. Also, the transitions have to be adapted which are located in the separate CSV file. The file contains a quadratic matrix consisting of the operations and transitions. Each operation needs a row with transitions to all operations. The transitions from the new operation require one additional LOC. Also, the other

rows are adapted to support a transition to the new operation. In total, a new operation requires *N+95* LOC changes.

11. **Change behavior:** To change the behavior of a single HTTP operation in the BenchFlow DSL, one LOC has to be changed. The reason is that all transitions of an operation are in one line. An operation already contains transitions to all operations, but the transitions can have a probability of 0 %.

    In the ContinuITy model, the behavior is a part of the WESSBAS model. An operation contains three LOC for each available transition. This includes two LOC for the target state and the transition probability, and one LOC of the think time. If an operation does not contain a transition, three additional LOC are required for each transition. To change a single transition, two LOC has to be changed.

    The transitions from the JMeter model are in the separate CSV file. The file contains a quadratic matrix consisting of the operations and transitions. Each operation has a row with transitions to all operations. To change the behavior of an operation, the respective row has to be changed. Therefore, one LOC has to be adjusted to change the behavior of a single operation.

12. **New parameter:** To add a new parameter to the BenchFlow DSL, two new LOC are necessary for the particular HTTP operation. For the definition of the parameter type, one LOC is required which is, in this case, the query parameter. The second LOC consists of the parameter name and the parameter value. If the HTTP operation already contains a query parameter, the first line is not required because the parameter type is already defined. In this case, one addition LOC is necessary.

    Using the ContinuITy model, a new parameter requires changes in the IDPA annotation and IDPA application model. The application model needs one LOC to define a parameter container if no parameter is available for a particular operation. Two more LOC are necessary to define the parameter name, the parameter type, and a reference to the annotation model. In the annotation model, three LOC are necessary to define the application model reference, a reference within the file, and a parameter property. The parameter value needs additional three LOC, but when the parameter value is already existing in another parameter, then the parameter can be reused which leads to no additional LOC. In total, nine LOC are required for a new parameter without reusing another parameter. At least five LOC are required. In this case, a parameter is existing for the operation, and another parameter can be reused in the annotation model.

    The JMeter model needs two LOC for the parameter type if it is not available. For the initialization of the parameter, two LOC are required plus two LOC for the parameter name and parameter value. In addition to that, the parameter properties consist of three LOC which are not necessary. In total, six LOC are

required for a new parameter, and four LOC for a new parameter if a parameter is already existing.

6.2.4. Discussion of Results

The comparison of the previous section shows that the BenchFlow DSL extensions have advantages and disadvantages compared to the ContinuITy and JMeter models. In the following, we will answer the RQ3. One advantage is, that regression conditions and deployment properties can be defined directly in the BenchFlow DSL so that all necessary information for a load test is in one file. The BenchFlow DSL contains the HTTP operations and the different behaviors for a set of SUT versions. However, ContinuITy needs at least three and JMeter needs at least two files to store all relevant load test information. Another advantage is that BenchFlow needs less or equal LOC changes to add new parameters and to change the behavior of operations in comparison to the other models.

The disadvantages of BenchFlow are described in the following. In the HTTP workload, the definition of an additional operation needs more LOC changes in comparison to the JMeter model. Furthermore, the change of the initial state can lead to more LOC changes in comparison to both other models. ContinuITy provides with the WESSBAS model a more dynamic solution because the behavior has not to be adapted when the operations or the order of the operations change. The behavior of the HTTP workload from BenchFlow is defined as a matrix containing a transition list for each operation. If the behavior for a specific transition should be changed, the position of the source and destination operation have to be determined so that it is possible to find the transition. In the ContinuITy and JMeter models, the transitions are defined by the operation keys which is easier to find a transition. JMeter provides the possibility to define regression conditions based on static values which not possible with the BenchFlow DSL. Another disadvantage is that the BenchFlow DSL does not support to define several IP addresses, e.g., for different service, because a global IP address is used.

The disadvantages are used to answer RQ4. The BenchFlow extensions can be improved to provide more flexibility for different fields of application. First of all, the initial state should be defined with an own parameter instead of using the first operation. This leads to less effort when the initial state will be changed. Regression conditions based on static values can be helpful to prove single SUT versions or to prove the upper bound of measurements. This can be realized with an extension of the envisioned regression conditions. The workload behavior should be a transition list with a quad tuple of (1) the source operation key, (2) the destination operation key, (3) the transition probabilities, and (4) the think times. In a transition list, the benefit is that few cells exist because

transitions with the probability of 0 % can be omitted. This leads to less effort when new operations are added or removed. To support several IP addresses, each HTTP operation can have an address parameter which overwrites the global address.

Chapter 7

Conclusion

This chapter summarizes the work of this thesis and discusses if the goal is reached. Section 7.1 presents the summary, and the discussion is covered in Section 7.2. Finally, Section 7.3 lists possible future works.

7.1. Summary

In this thesis, we developed an approach to execute automatically regression tests based on load testing with representative user behaviors using a declarative DSL. The tools BenchFlow and ContinuITy were extended for our approach. Our approach consists of three parts.

The first part is the extension of the declarative BenchFlow DSL to support the definition of HTTP workloads, regression-specific properties, and deployment properties. HTTP workloads contain information about HTTP operations and user behaviors. This information is necessary to execute load tests. The regression-specific properties contain the regression condition and the regression goals in order to support the regression analysis. The approach supports several regression goals. They differ in the used HTTP operations and user behaviors. The deployment properties contain information which is required for the deployment of the SUT, e.g., its Docker image.

The next part is the workflow of ContinuITy. ContinuITy provides the possibility to request HTTP workload with a representative user behavior. The HTTP workload is necessary for BenchFlow to execute the load tests and afterwards the regression analysis. A service calls the ContinuITy workflow with a BenchFlow DSL, whereby the workflow distinguishes between the different regression goals and handles the ContinuITy models with the representative user behaviors differently. The workflow contains a transformation which transforms the HTTP workloads from the ContinuITy

models into BenchFlow workload models. Afterwards, the BenchFlow workload model is added to the provided BenchFlow DSL.

The BenchFlow DSL used within the BenchFlow workflow is the last part of our approach. The BenchFlow workload models are transformed to JMeter test plans so that JMeter can execute load tests. Based on the deployment properties, different SUT versions are deployed that are load tested with JMeter. The results of the load tests are used for the regression analysis to detect regressions.

In our evaluation, we evaluated the BenchFlow DSL extensions and the regression analysis. The regression analysis was evaluated with different considered scenarios. Two different applications were used in scenarios, one sample application and one test application to provide synthetic measurements. The results of the scenarios were compared against our expected results, and based on that we answered our research questions. Our evaluation shows that the regression analysis can correctly detect regressions in applications, which are tested with and without corresponding representative user behaviors. For the evaluation of the extensions, features were considered which were used to compare the BenchFlow DSL with related approaches. With the results of the comparison, the research questions were answered. Furthermore, the results of our second evaluation show that the effort to make a change in the BenchFlow DSL can be in some cases high and in other cases that the effort can be low. The effort depends on the design of the BenchFlow DSL extensions.

7.2. Retrospective

The aim of this thesis was divided into five different goals which were mentioned in Section 1.3. In this section, we want to discuss whether the goals were reached.

The first goal was to extend the BenchFlow DSL to support HTTP workloads, regression-specific properties, and deployment properties. These extensions are explained in Section 4.2. Section 4.2.2 describes the regression goals and the regression conditions.

The second goal was to transform the ContinuITy model into the BenchFlow DSL. The ContinuITy model consists of three different models which have to be selected before the transformation. This is a part of the Section 4.3.3, and the corresponding transformation into the BenchFlow workload model is explained in Section 4.3.4. The complete workflow to receive a BenchFlow DSL is described in Section 4.3.

The third goal was to support a regression analysis in BenchFlow. The regression analysis supports the detection of regression based on load test measurements. It is described

in Section 4.4.2. The regression analysis is a part of the BenchFlow workflow which is explained in Section 4.4.

The fourth goal was to evaluate the regression analysis from the fourth goal. The evaluation is described in Section 6.1, and contains different scenarios with two different applications for the SUT which are mentioned in Section 6.1.2. The different applications provide real and synthetic measurements to verify whether the regression analysis detects regressions correctly. The analysis detects false regression, but the analysis mainly works correctly. The discussion of the results are part of Sections 6.1.4 and 6.1.5. Finally, the threats to validity of this evaluation are explained in Section 6.1.6.

The last goal was to evaluate the BenchFlow DSL extensions from the first goal. The evaluation is part of Section 6.2 including the research questions in Section 6.2.1. In Section 6.2.2, features are considered which were used for the comparison with related models. The discussion of the results and the answers to the research questions are part of Section 6.2.4. This includes possible improvements of the BenchFlow extensions.

The aim of this thesis was to provide an approach for automated regression testing based on load testing with representative user behaviors using a declarative approach. We reached the aim because all goals were reached.

7.3. Future Work

This section describes possible future works which are out of the scope of this thesis.

The first possible future work is the improvement of the declarative BenchFlow DSL or rather the extensions of the BenchFlow DSL to support more fields of application. As discussed during the thesis, the extensions of BenchFlow have some limitations. Limitations are (1) that neither global operations nor global data sources can be used, (2) that the sequential retrieval of items in a list or rows in a CSV file is not supported, and (3) that only one regression condition is supported as mentioned in Section 4.2.2.2. The improvements of the limitations can be combined with possible improvements from the evaluation in Section 6.2.4. The possible improvements from the evaluation are (1) that the initial state should be defined with an own parameter instead of using the first operation and (2) that a regression condition can be defined to prove the upper bound of measurements. In addition to that, (3) the workload behavior should be a transition list with a quad tuple of the source operation key, the destination operation key, the transition probabilities, and the think times. Finally, (4) each HTTP operation should have an IP address field to overwrite the global IP address so that several services can be regression tested.

The second possible future work is based on the results of the second evaluation or more precisely on the threats to validity in Section 6.1.6. This work aims to provide more measurements for operations in SUT versions which contain a too low probability for a call which leads to zero or only a few measurements. The total number of measurements can be increased by increasing the load test duration, but this does not lead to more measurements for these operations automatically. A solution can be an adaption of the user behavior to increase the probabilities of these operations. This adaption has to be as low as possible so that the user behavior remains representative. This solution requires a consideration of the load test duration, the change of the representative user behavior, and the possibility to detect regressions.

The sample application which was used for the evaluation consists of around 50 operations and was not distributed. Therefore, the regression analysis should be tested with measurements of applications which are distributed over different servers and systems, and which consist of more than 50 operations. That can be combined with more users and a longer duration of the load tests because these applications contain more operations and therefore a lower probability that all operations are called.

Appendix A

Appendix

A.1. WESSBAS DSL

Listing A.1 shows an example of a WESSBAS DSL.

A.2. Behavior Model

Listing A.2 shows an example of a behavior model in JSON format.

A.3. BenchFlow DSL

Listing A.3 shows an example of a BenchFlow DSL without workload in YAML format. The corresponding workloads are displayed in Listings A.4 and A.5.

A.4. Regression Report

Figure A.1 shows a page from a regression report. It consists of two different SUT versions, a detected regression of the operation *get* and two plots for the visualization of the elapsed time measurements of the operation *get*.

Listing A.1 A simplified example of a WESSBAS DSL.

```
<m4jdsl:WorkloadModel ...>
  <applicationModel>
    <sessionLayerEFSM initialState="ASId1_shopUsingGET">
      <applicationStates eId="ASId1_shopUsingGET">
        <service name="shopUsingGET"/>
        ...
        <request xsi:type="m4jdsl:HTTPRequest" eId="R1 (shopUsingGET)">
          <properties key="HTTPSampler.domain" value="127.0.0.1"/>
          <properties key="HTTPSampler.port" value="8080"/>
          <properties key="HTTPSampler.path" value="/shop"/>
          <properties key="HTTPSampler.method" value="GET"/>
          <properties key="HTTPSampler.protocol" value="http"/>
        </request>
        ...
      </applicationStates>
      <applicationStates eId="ASId2_addUsingPOST">
        ...
      </applicationStates>
      <exitState eId="$"/>
    </sessionLayerEFSM>
  </applicationModel>

  <behaviorMix>
    <relativeFrequencies behaviorModel="//@behaviorModels.1" value="0.75"/>
    <relativeFrequencies behaviorModel="//@behaviorModels.0" value="0.25"/>
  </behaviorMix>

  <behaviorModels name="gen_behavior_model0" initialState="MSId2_shopUsingGET">
    <markovStates eId="MSId2_shopUsingGET" ...>
      <outgoingTransitions targetState="MSId3_addUsingPOST" probability="0.84">
        <thinkTime xsi:type="m4jdsl:NormallyDistributedThinkTime" mean="543.0"/>
      </outgoingTransitions>
      <outgoingTransitions targetState="MSId1" probability="0.16">
        <thinkTime xsi:type="m4jdsl:NormallyDistributedThinkTime"/>
      </outgoingTransitions>
    </markovStates>
    <markovStates eId="MSId3_addUsingPOST" ...>
      ...
    </markovStates>
    <exitState eId="MSId1"/>
  </behaviorModels>
  <behaviorModels ...>
    ...
  </behaviorModels>
</m4jdsl:WorkloadModel>
```

Listing A.2 An example of a behavior model.

```
{
    "behaviors": [{
        "name": "gen_behavior_model1",
        "initialState": "CartGET",
        "probability": 0.48,
        "markov-states": [{
            "id": "CartGET",
            "transitions": [{
                "targetState": "ShopPOST",
                "probability": 1,
                "think-time-mean": 0,
                "think-time-deviation": 0
            }]
        },{
            "id": "ShopPOST"
        }]
    },{
        "name": "gen_behavior_model0",
        "initialState": "CartGET",
        "probability": 0.52,
        "markov-states": [{
            "id": "CartGET",
            "transitions": [{
                "targetState": "ShopPOST",
                "probability": 0.75,
                "think-time-mean": 2157,
                "think-time-deviation": 0
            }]
        },{
            "id": "ShopPOST",
            "transitions": [{
                "targetState": "CartGET",
                "probability": 0.6,
                "think-time-mean": 2124,
                "think-time-deviation": 127
            },{
                "targetState": "ShopPOST",
                "probability": 0.2,
                "think-time-mean": 3210,
                "think-time-deviation": 0
            }]
        }]
    }]
}
```

Listing A.3 An example of a BenchFlow DSL without workload.

```
version: '1'
name: test-v1-v2
sut:
  name: test-service
  version:
   - '1'
   - '2'
  type: http
  configuration:
    target_service:
      name: test-service
      endpoint: localhost:8080
    deployment:
      docker_image: test-service
      docker_image_port_binding: 8080:8080
      docker_host_port: '2375'
      ping_url_path: /swagger-ui.html

configuration:
  users: 10
  goal:
    type: INDIVIDUAL_REGRESSION
  workload_execution:
    ramp_up: 1m
    steady_state: 1m # Not used but mandatory
    ramp_down: 1m   # Not used but mandatory
  termination_criteria:
    test:
      max_time: 30m
    experiment:     # Not used but mandatory
      type: FIXED
      number_of_trials: 1
  settings:         # Not used but mandatory
    stored_knowledge: false
  quality_gates:
    test:
      regression:
        metric: median-response-time@@operation
        delta_absolute: 50ms
        delta_percent: 25%
data_collection:    # Not used but mandatory
  client_side:
    faban:
      max_run_time: 1m
      interval: 1m

workload: ...
```

Listing A.4 An example of a BenchFlow HTTP workload.

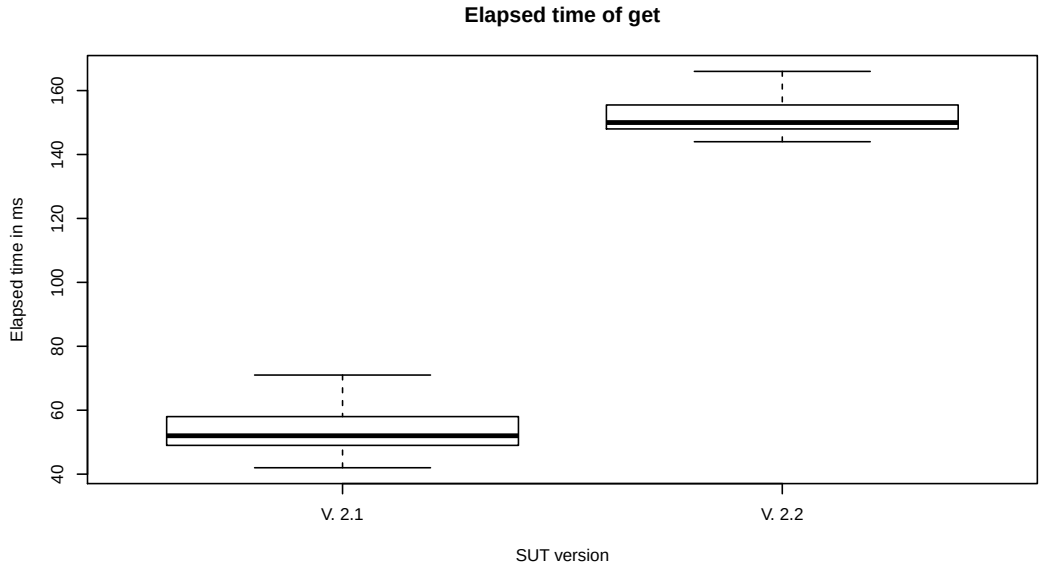
```
first_test_workload:
  sut-version: '1'
    workload-items:
      basisWorkload:
        driver_type: HTTP
        data-sources:
         - path: C:\Temp\CSV\data.csv
           delimiter: ','
        operations:
         - id: set
           endpoint: /storage/set/${Language}
           method: POST
           protocol: http
           body:
             content:
              - Java
              - Scala
              - SQL
           extract-regexp:
             extract_id:
               pattern: (.*)
               default: 'NO VALUE'
               match-number: 0
         - id: get
           endpoint: /storage/get/${extract_id}
           method: GET
           protocol: http
        mix:
          matrix:
           - [25.0% tt(1000.0 500.0), 75.0% tt(2000.0 400.0)]
           - [20.0% tt(1000.0 500.0), 50.0% tt(2000.0 200.0)]
```

Listing A.5 An example of a BenchFlow HTTP workload.

```
second_workload_test:
  sut-version: '2'
  workload-items:
    basisWorkload:
      driver_type: HTTP
      operations:
       - id: set
         endpoint: /storage/set/${id}
         method: POST
         protocol: http
         url-parameter:
           id: my-language
         body:
           content:
            - Java
            - JavaScript
         extract-regexp:
           extract_id:
             pattern: (.*)
             default: 'NO VALUE'
       - id: get
         endpoint: /storage/get/${extract_id}
         method: GET
         protocol: http
       - id: search
         endpoint: /storage/search
         method: GET
         protocol: http
         query-parameter:
           content:
            - Java
            - JavaScript
         extract-jsonpath:
           matched_ids:
             pattern: $.data.ids
             default: '0'
       - id: getAll
         endpoint: /storage/all/${matched_ids}
         method: GET
         protocol: http
      mix:
        matrix:
         - [25.0%, 75.0%, 0.0%, 0.0%]
         - [20.0%, 50.0%, 30.0%, 0.0%]
         - [0.0%, 0.0%, 10.0%, 90.0%]
         - [0.0%, 0.0%, 0.0%, 0.0%]
```

Detected regressions

Metric	Old version	New version	Old value	New value	Abs. delta	Rel. delta
median−response−time	test−application−2.1	test−application−2.2	50	150	10	



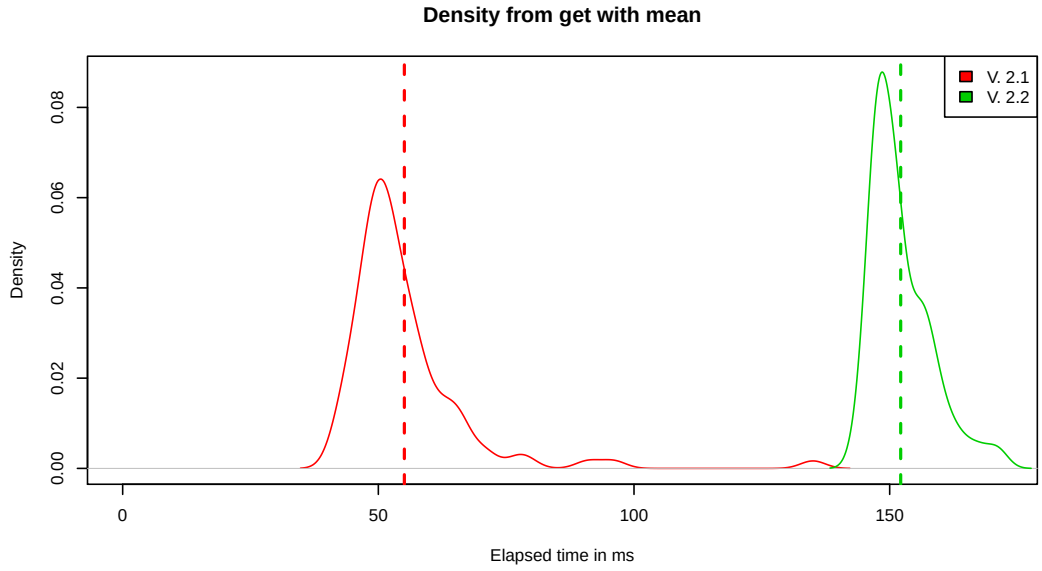


Figure A.1.: Page from a regression report with a detected regression in an operation.

Appendix A

Bibliography

[ASP18] The Apache Software Foundation. *Apache JMeter*. 2018. URL: http://jmeter.apache.org/ (cit. on p. 76).

[BK17] A. Brunnert, H. Krcmar. "Continuous performance evaluation and capacity planning using resource profiles for enterprise applications." In: *Journal of Systems and Software* 123 (2017), pp. 239–262 (cit. on p. 18).

[Bla18] BlazeMeter. *Taurus: Automation-friendly framework for Continuous Testing*. 2018. URL: http://gettaurus.org/ (cit. on pp. 17, 22).

[BPV+16] M. Blohm, M. Pahlberg, S. Vogel, J. Walter, D. Okanovic. "Kieker4DQL: Declarative performance measurement." In: *Proceedings of the 2016 Symposium on Software Performance (SSP)*. 2016 (cit. on p. 16).

[DAv16] S. D'Avico. "The BenchFlow Framework for Automated Performance Experiments Execution on Heterogeneous Middleware Systems." MA thesis. Università della Svizzera Italiana, Sept. 2016 (cit. on p. 21).

[Fin17] J. Findahl. "Automating Goal-Driven Performance Tests in BenchFlow." MA thesis. Università della Svizzera Italiana, Sept. 2017 (cit. on pp. 9, 23, 24, 27).

[FJA+10] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, P. Flora. "Mining performance regression testing repositories for automated performance analysis." In: *Quality Software (QSIC), 2010 10th International Conference on*. IEEE. 2010, pp. 32–41 (cit. on pp. 15, 17, 18).

[FP17] V. Ferme, C. Pautasso. *BenchFlow - A Platform for End-to-end Automation of Performance Testing and Analysis - Presentation*. 2017 (cit. on pp. 2, 9).

[FP18] V. Ferme, C. Pautasso. “A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments.” In: *9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*. ACM. Berlin, Germany: ACM, Apr. 2018 (cit. on pp. 9, 28).

[GBK14] F. Gorsler, F. Brosig, S. Kounev. “Performance queries for architecture-level performance models.” In: *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*. ACM. 2014, pp. 99–110 (cit. on p. 16).

[GWP+16] S. Ghaith, M. Wang, P. Perry, Z. M. Jiang, P. O’Sullivan, J. Murphy. “Anomaly detection in performance regression testing by transaction profile estimation.” In: *Software Testing, Verification and Reliability* 26.1 (2016), pp. 4–39 (cit. on pp. 15, 18).

[GWPM13] S. Ghaith, M. Wang, P. Perry, J. Murphy. “Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems.” In: *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE. 2013, pp. 379–383 (cit. on pp. 15, 18).

[HB10] J. Heer, M. Bostock. “Declarative language design for interactive visualization.” In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (2010), pp. 1149–1156 (cit. on pp. 2, 7).

[HF10] J. Humble, D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010 (cit. on p. 1).

[HHF13] C. Heger, J. Happe, R. Farahbod. “Automated root cause isolation of performance regressions during software development.” In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM. 2013, pp. 27–38 (cit. on p. 18).

[HHMO17] C. Heger, A. van Hoorn, M. Mann, D. Okanović. “Application performance management: State of the art and challenges for the future.” In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM. 2017, pp. 429–432 (cit. on p. 11).

[ISO 90003] *ISO/IEC 90003:2014-12, Software engineering - Guidelines for the application of ISO 9001:2008 to computer software*. Standard. Dec. 2014 (cit. on p. 6).

[JH15] Z. M. Jiang, A. E. Hassan. “A survey on load testing of large-scale software systems.” In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1091–1118 (cit. on p. 1).

[JHHF09] Z. M. Jiang, A. E. Hassan, G. Hamann, P. Flora. “Automated performance analysis of load tests.” In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE. 2009, pp. 125–134 (cit. on pp. 17, 18).

[JWS+18] I. Jimenez, N. Watkins, M. Sevilla, J. Lofstead, C. Maltzahn. “quiho: Automated Performance Regression Testing Using Inferred Resource Utilization Profiles.” In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM. 2018, pp. 273–284 (cit. on p. 17).

[KJMG17] R. Kazmi, D. N. Jawawi, R. Mohamad, I. Ghani. “Effective regression test case selection: a systematic literature review.” In: *ACM Computing Surveys (CSUR)* 50.2 (2017), p. 29 (cit. on p. 17).

[LCL12] D. Lee, S. K. Cha, A. H. Lee. “A performance anomaly detection and analysis framework for DBMS development.” In: *IEEE Transactions on Knowledge and Data Engineering* 24.8 (2012), pp. 1345–1360 (cit. on p. 18).

[Lin06] G. Linden. *Make Data Useful*. 2006. URL: https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt (cit. on p. 1).

[MFB+07] J. Meier, C. Farre, P. Bansode, S. Barber, D. Rea. *Performance testing guidance for web applications: patterns & practices*. Microsoft press, 2007 (cit. on pp. 5, 6).

[MT12] S. P. Meyn, R. L. Tweedie. *Markov chains and stochastic stability*. Springer Science & Business Media, 2012 (cit. on p. 8).

[NTRSS18] NovaTec Consulting GmbH and University of Stuttgart (Reliable Software Systems Group). *ContinuITy Research Project*. 2018. URL: https://continuity-project.github.io/ (cit. on p. 10).

[PHG14] M. Pradel, M. Huggler, T. R. Gross. “Performance regression testing of concurrent classes.” In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM. 2014, pp. 13–25 (cit. on p. 18).

[PV06] M. Pesic, W. M. Van der Aalst. “A declarative approach for flexible business processes management.” In: *International conference on business process management*. Springer. 2006, pp. 169–180 (cit. on p. 7).

[RUCH01] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold. “Prioritizing test cases for regression testing.” In: *IEEE Transactions on software engineering* 27.10 (2001), pp. 929–948 (cit. on p. 17).

[SAH18] H. Schulz, T. Angerstein, A. van Hoorn. “Towards Automating Representative Load Testing in Continuous Software Engineering.” In: 2018 (cit. on pp. 1, 10).

[San17] W. Santos. *Which API Types and Architectural Styles are Most Used?* 2017. URL: https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26 (cit. on p. 1).

[SHNF15] W. Shang, A. E. Hassan, M. Nasser, P. Flora. "Automated detection of performance regressions using regression models on clustered performance counters." In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM. 2015, pp. 15–26 (cit. on pp. 15, 18).

[Sub06] B. Subraya. *Integrated Approach to Web Performance Testing: A Practitioner's Guide: A Practitioner's Guide*. IGI Global, 2006 (cit. on pp. 5, 6).

[VHH18] F. Van den Berg, J. Hooman, B. R. Haverkort. "A Domain-Specific Language and Toolchain for Performance Evaluation Based on Measurements." In: *International Conference on Measurement, Modelling and Evaluation of Computing Systems*. Springer. 2018, pp. 295–301 (cit. on p. 16).

[VHS+18] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, H. Krcmar. "WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems." In: *Software & Systems Modeling* 17.2 (May 2018), pp. 443–477 (cit. on pp. 7, 11).

[VKV00] A. Van Deursen, P. Klint, J. Visser. "Domain-specific languages: An annotated bibliography." In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36 (cit. on pp. 7, 16).

[VRH14] F. Van den Berg, A. Remke, B. R. Haverkort. "A domain specific language for performance evaluation of medical imaging systems." In: *OASIcs-OpenAccess Series in Informatics*. Vol. 36. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2014 (cit. on p. 16).

[VWH12] A. Van Hoorn, J. Waller, W. Hasselbring. "Kieker: A framework for application performance monitoring and dynamic software analysis." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM. 2012, pp. 247–248 (cit. on pp. 11, 16).

[WEG+18] J. Walter, S. Eismann, J. Grohmann, D. Okanovic, S. Kounev. "Tools for Declarative Performance Engineering." In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM. 2018, pp. 53–56 (cit. on p. 16).

[WHK+16] J. Walter, A. van Hoorn, H. Koziolek, D. Okanovic, S. Kounev. “Asking What?, Automating the How?: The Vision of Declarative Performance Engineering.” In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM. 2016, pp. 91–94 (cit. on pp. 7, 16).

[WOK17] J. Walter, D. Okanović, S. Kounev. “Mapping of Service Level Objectives to Performance Queries.” In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM. 2017, pp. 197–202 (cit. on p. 16).

[Wor11] S. Work. *How Loading Time Affects Your Bottom Line*. 2011. URL: https://blog.kissmetrics.com/loading-time/ (cit. on p. 1).

[WW17] J. Wienke, S. Wrede. “Performance regression testing and run-time verification of components in robotics systems.” In: *Advanced Robotics* 31.22 (2017), pp. 1177–1192 (cit. on p. 18).

[YH12] S. Yoo, M. Harman. “Regression testing minimization, selection and prioritization: a survey.” In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120 (cit. on p. 17).

All links were last followed on December 20, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature