PLeTsPerf - A Model-Based Performance Testing Tool

Elder M. Rodrigues, Maicon Bernardino, Leandro T. Costa, Avelino F. Zorzo, Flávio M. Oliveira
School of Computer Science
Pontifical Catholic University of Rio Grande do Sul - PUCRS
Porto Alegre - Rio Grande do Sul - Brazil
Email:{elder.rodrigues, avelino.zorzo, flavio.oliveira}@pucrs.br
bernardino@acm.org, leandro.teodoro@acad.pucrs.br

***Abstract*—Performance testing is a highly specialized task, since it requires that a performance engineer knows the application to be tested, its usage profile, and the infrastructure where it will execute. Moreover, it requires that testing teams expend a considerable effort and time on its automation. In this paper, we present the PLeTsPerf, a model-based performance testing tool to support the automatic generation of scenarios and scripts from application models. PLetsPerf is a mature tool, developed in collaboration with an IT company, which has been used in several works, experimental studies and pilot studies. We present an example of use to demonstrate the process of generating test scripts and scenarios from UML models to test a Web application. We also present the lessons learned and discuss our conclusions about the use of the tool.**

Keywords*—*Performance Testing, Model-Based Testing, Testing Tool.

I. Introduction

It is well-known that the testing phase is one of the most important and time consuming phases of a software development project [1]. Software quality is a very important asset, for this reason even small and simple projects must be carefully tested during Unit, Integration, System and Acceptance testing levels [2]. The main idea is to detect software failures or non-functional issues in all testing levels. Therefore, the use of a tool supporting the testing process is crucial. In the last decades, several testing tools were developed to support one or several testing levels [3] [4]. For instance, System testing normally requires some highly skilled professionals and a testing tool to support each one of the system testing types, *i.e.*, Functional, Security or Performance [5].

Performance testing is a highly specialized task. It involves knowledge about the application to be tested, its usage profile, and the infrastructure where it will operate. Furthermore, because it involves intensive test automation, performance testing requires understanding of the performance test automation tools that will be used. Hence, there is a bottleneck in productivity of performance engineering teams, due to the increasing complexity of the performance testing tools and the workload to generate test cases or scripts. Moreover, most of the performance testing tools are based on the Capture & Replay (CR) technique.

Although this technique has brought several benefits, since it allows to design performance scripts through recording the user interactions with the System Under Test (SUT), it also has some limitations. For example, when a performance engineer makes some mistake during the recording of a test case, it is necessary to restart and record all user interactions from the beginning. Moreover, in case of a change in the application, motivated by a change in a requirement or to fix a "bug", the scripts developed to test this requirement must be re-written from scratch. Furthermore, if new test engineers are hired, they will have a steep learning curve to understand the scripts. Another concern is related to the need of manually setting all scenario information, *e.g.* ramp up, ramp down, number of Virtual Users (Vusers), etc.

In order to mitigate the above drawbacks and to improve communication between development and testing teams, we began development on a model-based performance tool. Moreover, adopting a Model-Based Testing (MBT) approach allows performance teams to get involved in designing the test models from early stages of the software development process.

This paper is organized as follows. Section II presents some background about performance testing, introduces some related work and describes the context of the development of our tool. In Section III we present a process to apply model-based performance testing, as well as to describe in details our MBT tool. Section IV demonstrates how to apply our approach and tool to generate scripts and scenarios from UML Use Case and Activity Diagrams. Section V points out the lessons we have learned from the development of this work. Finally, in Section VI we discuss our conclusions.

II. Background

In this section, we provide background information on Model-Based Testing (MBT) and Performance Testing. We also discuss some model-based tools to apply performance testing and the context in which our study was developed.

A. A Brief Overview

Performance testing is a superset encompassing some subcategories of testing, such as: load testing, stress testing, endurance testing, or capacity testing [6]. During the performance testing a System Under Test (SUT) is submitted to a synthetic workload into a managed test environment. This synthetic workload is powered by a test model, which is generated in accordance with the testing requirements. This model is specified with system functionalities, test data and a set of workload attributes such as number of users. The goal is to

978-1-4799-7125-1/15/$31.00 ©2015 IEEE

measure and evaluate the performance metrics, *e.g* response times, throughput and resource utilization levels. Based on the model, a test analyst sets the load generators to mimic the behavior of real system users, called virtual users (Vusers) [7].

There are several tools to support, with different degrees of automation, the generation of a synthetic workload. On one hand, some tools require that a performance analyst design the test cases and scenarios based on the testing requirements. On the other hand, some tools support the design of a test model, from which the synthetic workload is generated. Usually, these tools are developed based on one of the following techniques: Capture & Replay (CR) or Model-Based Testing (MBT).

A tool based on the CR technique runs in a "record" mode while the tester interacts manually with the SUT, based on the test requirements. After that, the tester configures the test scenario and defines a set of test parameters and metrics, such as test duration, users ramp up/down, total number of Vusers and response time. After the parameters are set, the tool is executed in a "workload" mode, the Vusers are created and the synthetic workload is generated.

MBT is a technique that takes advantage of the applications models to generate testing models suitable for the generation of test artifacts. Reusing or creating applications models to represent an abstraction of its behaviour is an useful way to inspect quality properties and performance attributes, since it promotes the understanding of the application and enables the testing design to be performed concurrently with the application development [8]. Thus, to take advantage of an MBT approach, a performance engineer must design or reuse some system models, add performance test information (*e.g.* number of Vusers and application parameters) to them and then load the model into an MBT tool for the generation of the test scenarios and scripts.

In the last decades, the CR technique has been used by several industrial performance testing tools such as Apache JMeter[1], MS Visual Studio[2], IBM Rational Performance Tester[3], HP LoadRunner[4] and Borland Silk Performer[5]. On the other hand, only in the last years the use of the MBT technique to generate performance scripts started to be explored by some tools such as WALTy [9], SWAT [10] and TestOptimal [4].

B. Related Work

Despite the amount of literature on performance testing, model-based performance testing tools are quite limited. In this section, we summarize some of the tools and approaches that have been developed and that are related to our proposed tool. We investigated some tools that apply MBT to support performance testing, such as WALTy [9], SWAT [10] and TestOptimal [4].

Ruffo *et al.* [9] proposed WALTy (Web Application Load-based Testing), a set of tools that allow performance analysis of Web applications. The tool usage is based on a workload characterization modeled using Customer Behavior Model Graphs (CBMG). These models are generated from information extracted from log files. Although relevant, this approach create the models from log-based files, which requires that the system is already developed. Ideally, the SUT models should be designed since the early software development phases and reusing previous designed models, *i.e.* UML models.

Shams *et al.* [10] designed an approach that uses a modified version of the SWAT (Session-Based Web Application Tester) tool. The tool uses an application model, modeled using Extended Finite State Machine (EFSM), which captures the application behaviour amd generates the test cases. To create the syntetic workload the tool is integrated to `httperf` loaf generator tool. PLeTsPerf and SWAT adopted a different strategy to generate load. While SWAT is integrated into the `httperf` load generator, our tool was designed to work as a front-end to several commercial load generators, *i.e.* Load Runner or Visual Studio.

TestOptimal [4] is an integrated test design and test automation tool to support MBT. TestOptimal has a suite of MBT tools for both functional and performance testing. Its solution combines MBT and Data-Driven Testing (DDT) to perform test case generation and test automation. Among its main features, we can highlight the support to: MBT modeling - EFSM, state diagram (UML), Control Flow Graph (CFG), and activity diagram (UML); Graphs - model graph, sequence graph, coverage graph and Message Sequence Chart (MSC), and; Test Case Generation - random walk, optimal sequencer, mutant path, priority path, mCase (custom test case) sequencer. Despite TestOptimal is a powerful tool, one of the most complete MBT tool available, it requires that a MBT adopter invest a reasonable amount of money in software licenses (it is a commercial tool) and training. These adoption costs can lead some companies to developed in-house customized solutions to apply performance model-based testing.

C. Context

PLeTsPerf was developed as part of our effort to automate the performance testing activities, in the context of a collaboration project between our research group on Software Testing[6] and a Technology Development Lab (hereafter referred as TDL) of a global IT company. Along the collaboration, our main research focus was to investigate innovative ways to mitigate the effort of creating test scenarios and scripts to apply performance testing. Thus, the target tool was designed to support testing teams on the adoption of the MBT technique [5] to generate performance test scenarios and scripts from SUT models. Therefore, the tool must take a SUT behavioural model as input, generate abstract test cases and scenarios and then produce performance scripts as output to a specific testing technology, *i.e.* LoadRunner (LR) [11]. Thus, PLeTsPerf was designed to work as a front end to this tool, generating scenarios and scripts that are used by the load generator tool to run the test.

An initially version of the tool, developed from scratch, based on a software product line, was released in 2011 [12]. That tool focused on the static generation of scripts and scenarios from UML models. A second version, focusing on the

[1] jmeter.apache.org

[2] www.visualstudio.com

[3] www.ibm.com/software/products/en/performance

[4] www8.hp.com/us/en/software-solutions/loadrunner-load-testing

[5] www.borland.com/Products/Software-Testing/Performance-Testing/Silk-Performer

[6] www.cepes.pucrs.br

generation of scripts and scenarios from abstract intermediate models, was released in 2012 [11]. To evaluate the tool we designed an experimental study in 2014. The aim of this study was to evaluate the effort to use a CR-based approach using LR and an MBT approach using PLeTsPerf [13]. The results of the study indicate that the use of our model-based tool reduces the effort required to test an application. To better investigate these results we set up, in collaboration with the TDL, other experimental study. In this study, we compare the use of our tool, to generate scripts and scenarios, with two Capture & Replay tools: HP LoadRunner and MS Visual Studio [14].

Motivated by the results of our experimental studies, our partner started a pilot study to evaluate the tool in a real testing project, and in case of success start an initial adoption of the tool by some of their performance testing teams.

III. PLeTsPerf - A Model-Based Performance Testing Tool

The aim of our tool is to enable a commercial performance testing tool take advantage of the MBT technique. Moreover, the PLeTsPerf can be extended to support the generation of scripts for several performance testing tools such as Jmeter and Visual Studio. However, in this paper we will only focus on the generation of scenarios and scripts to LoadRunner. Figure 1 depicts our model-based performance testing process. This process is comprised of eleven main activities, performed by three profiles according to role or tool: `Performance Engineer`, `PLeTsPerf`, and `Load Generator`. Some activities can read and/or write test artifacts, *e.g.* `Performance Scripts`. The details related to the activities of our testing process are described as follows:

1) Design the System Models - UML: `Design the System Models` is the first activity in our process. It consists of modeling the system requirement specifications (`Requirements` artifact) to generate the test artifacts. In this activity, UML diagrams are designed to support the later generation of scripts and scenarios. These models include test information to mimic the users behavior in the SUT. This activity is based on two broadly used UML diagrams: Use Case (UC) and Activity Diagram (AD). An UC diagram can be mapped to a performance test scenario, where each actor element represents a user profile and each use case element describes a complete system functionality performed by one or more actors. It is important to highlight that an use case element describes a detailed behavior, which is decomposed in the corresponding AD. This diagram is used to model dynamic interaction of the users with a system, representing it as a sequence of actions. Furthermore, one or several test cases could be derived from a single AD. Normally, an AD with a linear flow of actions will result in one test case. In case of an AD with alternative flows, several test cases can be generated. The number of flow is related to the number of decision node elements included in the AD. It is important to note that in our context this activity is performed by a performance engineer, since the developers and testers are not familiar with UML models, in special applying to performance testing. Furthermore, it could be useful if these models were designed by the business analyst or system analyst and then be delivered to the performance teams. Thus, the "cost" to design the model could be spread over all phases of the software development process and at the same time improve the quality of the system specification.

2) Add Performance Test Information: The next activity of our process consists of modeling performance test information in the UC and AD diagrams. For this purpose, we use performance stereotypes and tagged values, based on the SPT Profile [15], to annotate some specific UML elements with test information. The use of this approach allows us to describe the performance information necessary to generate test scenarios and scripts. The test information is annotated in fifty tagged values distributed into six stereotypes as follows:

(1) **PApopulation**: (a) *TDhost*: represents the hostname or IP address where the SUT is hosted; (b) *TDpopulation*: represents the number of Vusers that will interact with the SUT during the test; (c) *TDrampUpTime*: defines how long it takes for all Vusers to access the SUT; (d) *TDrampUpUser*: represents the rate at which the number of Vusers to access the SUT; (e) *TDrampDownTime*: defines how long it takes for all Vusers leave the SUT; (f) *TDrampDownUser*: represents the rate at which the number of Vusers leave the SUT; (2) **PAprob**: (a) *TDprob*: sets the number of user, defined as a percentage of the total number of users that will execute an existing activity; (3) **PAtime**: (a) *TDtime*: total time to perform a given performance test scenario, including the ramp up time, test execution time and ramp down time; (4) **PAthinkTime** (a) *TDthinkTime*: denotes the time between the moment the activity becomes available to the user and the moment the user decides to execute it, *i.e.*, the time spent by a user filling a form till its submission; (5) **PAaction**: (a) *TDaction*: defines the HTTP address (URL) used to submit input data by a user interaction; (b) *TDbody*: represents the body request; (c) *TDmethod*: defines the form submission method: POST or GET (default: POST); (d) *TDreferer*: represents the URL of the referring Web page, *i.e.* a page referred to the current page; (6) **PAparameters**: (a) *TDparameters*: represents the list of data fields used to define the form submission; (b) *TDsaveParameters*: saves dynamic data, as a parameter, to be used later.

After designing the model and annotating it with performance information, the performance engineer must export[7] the models to an `XMI file`, which is the input to the next activity of our process.

3) Parse Performance Test Model: In this activity, which is performed using PLeTsPerf, the performance test information modeled in the UML diagrams (contained in the `XMI file`) are parsed to a specific `Performance Structure` in memory. The test data stored in this structure will be used in the next steps of our process.

4) Generate Abstract Test Cases & Scenarios: The test data stored in the `Performance Structure` are used to generate abstract test cases and scenarios in a technology-independent format. For this purpose, this activity uses the test data to define the user behaviour, their current data inputs as well as the test scenario in a format that could be ease to understand by the tester. Thus, each abstract test case represents a sequence of actions performed by the Vusers, described in high level language. This activity also resolves the alternative flows to generate a set of abstract test cases (for more details see Section IV).

[7]Most of the UML modeling environments export models to an XMI file.

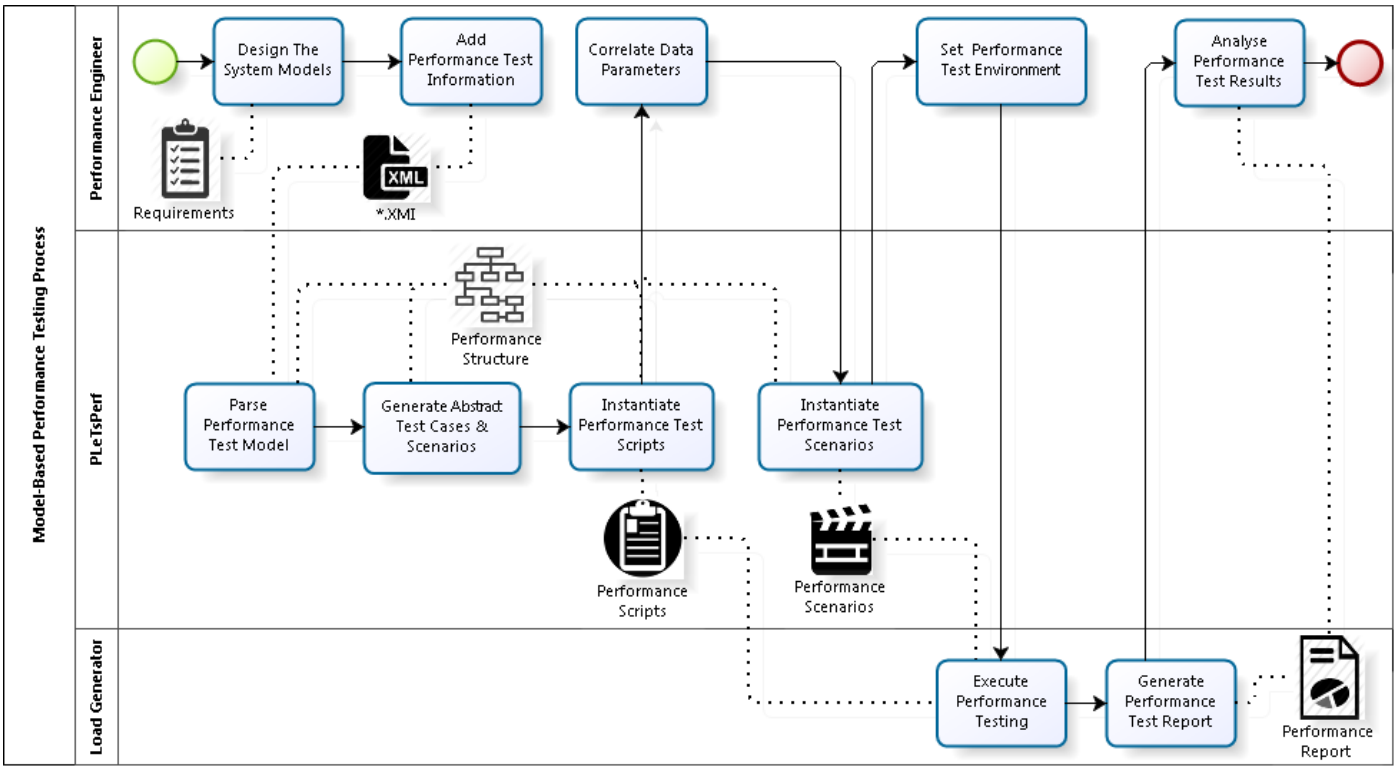

Fig. 1. Model-Based Performance Testing Process

5) Instantiate Performance Test Scripts: In this activity, the generated abstract test cases are instantiated to a specific performance testing technology, *i.e.*, LoadRunner scripts. Thus, the abstract test cases are bounded to a set of test parameters. Therefore, this activity generates as output `Performance Scripts` that will be used to execute the performance testing.

6) Correlate Data Parameters: In some applications there are parameters that are dynamically generated, *e.g.* Session ID, that is created by the server for security purpose. This type of parameter must be correlated after the test case generation. Besides, the tester must retrieve or generate this data or in some cases there are some applications providers that make available a list of correlated data parameters for its application, which facilitates the generation of performance test scripts. Thus, in this activity the performance engineer manually selects data for each parameter identified in the generated scripts.

7) Instantiate Performance Test Scenarios: At this point, it is possible to instantiate the performance test scenarios to a specific technology. These test artifacts are composed by different user profiles, in which a user profile can be composed by one or more Vusers. Each user profile can execute a set of performance test scripts. Moreover, additional details about the population can be set up, such as number of Vusers, ramp up, ramp down or pacing. The main purpose of the performance test scenarios is to mimic the synthetic workload of overall virtual users behavior as close as possible to "real" workload usage. As aforementioned, this activity generates, as result, a `Performance Scenarios` artifact, which together with `Performance Scripts` will be used to execute the performance test.

8) Set Performance Test Environment: In this activity, the performance engineer must set the performance test environment, such as, install the operating system, drivers and the SUT. Furthermore, they must set all related environment, such as databases and web servers, third-party applications, etc. Moreover, it is necessary to configure the workload generator and the monitoring module, as well as configure their environment.

9) Execute Performance Testing: In our approach, the PLeTsPerf works as a frond-end to the LoadRunner performance testing tool. Our MBT tool provides the generation of scenarios and scripts from the SUT models. Therefore, in order to execute the test, our tool just makes an internal call to the `Load Generator` (LG) and then loads the scripts and scenarios into the tool. After that, the LG is responsible to run the test while monitoring the results.

10) Generate Performance Test Report: After the test execution, the monitoring module summarize the data collected during the test. For example, the data that are related to the monitored metrics and the infrastructure resources, *e.g.* CPU and bandwidth. The data results could be consolidated in a document called `Performance Report` that presents the test results through graphs and tables.

11) Analyse Performance Test Results: Finally, the last activity of our process is performed by the performance engineer. They are responsible to analyse `Performance Report` and verify whether performance metrics are within the established limits or not. For example, check if thresholds have been violated or the required information necessary to suggest a system optimization were collected.

IV. Example of Use

In this section, we apply our approach to test a Web application, Workforce Planning: Skill Management Tool (hereafter referred as Skills), that manages skills, certifications and experiences of employees for a given company. This application was developed in the context of a collaboration project between a research group of our university and a global IT company. Skills was implemented using Java, MySQL and Tomcat. Table I summarizes the main functionalities of Skills, which are used to explain our approach.

TABLE I. Functional requirements of Skills application

ID	Requirement	Description
RF01	Skills Management	The system should allow users to add and/or edit their skills on techniques, methodologies, technologies, software, hardware, or languages, setting the proficiency level, acquired date, and last year that they have used a skill.
RF02	Certifications Management	The system should enable users to add and/or edit their certifications, whether academic or professional. Furthermore, it is possible to set the provider company, attainment date, and additional comments for each certification.
RF03	Experiences Management	The system should allow users to register their experiences in several business areas. For this requirement, additional information can be provided: company name, industry group, area, role, role level, begin and end date.
RF04	Visualize Profile	The system should enable users to see their profiles regarding skills, certifications or experience from registered data.

A. UML Diagrams

Based on the Skills functional requirements presented in Table I, the performance engineer must design the testing models, *i.e.*, UC and AD[8]. Thus, an UC model that represents the test scenario, has four use case elements, one for each application requirement. In turn, each one of these use cases is composed by one or more test cases. The design of the application testing models are as follows:

1) Use Case Design: In the UC diagram each actor element represents a user profile (see Figure 2). To represent the Skills user profiles two actors were modeled. For each actor three stereotypes were attributed: `PApopulation`, `PAtime` and `PAprob`. Basically, the profiles differ in the following aspects:

Headhunter: represents a human resource manager, which is responsible to assign employees to a software project according to their profiles. This actor element represents fifty Vusers. At the beginning of the test, ten Vusers (`TDrampUpUser`) are added and more ten are added per minute (`TDrampUpTime`) during four minutes. At the end of the test, ten Vusers (`TDrampDownUser`) per minute (`TDrampDownTime`) leave the system. Therefore, in order to perform the ramp down and ramp up, ten minutes are added to the test execution time (1h50min), which results in a total test execution (`TDtime`) of two hours, that corresponds to: ramp up + ramp down + test execution;

Employee: represents a company employee, which accesses the system to inform skills and technical certifications. This actor element represents nine hundred fifty Vusers. At the beginning of the test, fifty Vusers (`TDrampUpUser`) are added and one hundred are added each ten minutes (`TDrampUpTime`) during ninety minutes. At the end of the test, ten Vusers (`TDrampDownUser`) per minute (`TDrampDownTime`) leave the system. Therefore, in order to perform the ramp down and ramp up, 90 minutes are added to the test execution time (3h), which results in a total test execution (`TDtime`) of four hours (14400s), that corresponds to: ramp up + ramp down + test execution;

[8]A sample of the models and scripts designed when performing the example of use can be found in: http://www.cepes.pucrs.br/tool/pletsperf

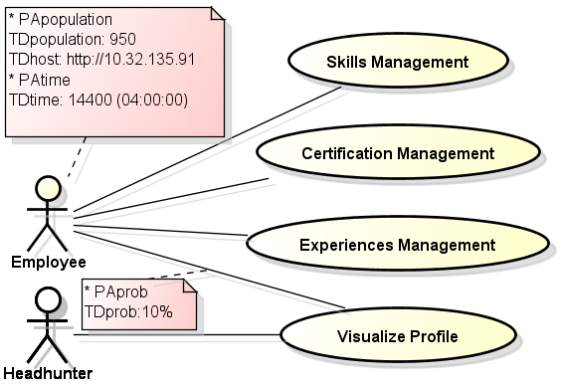

Fig. 2. Skills Use Case Diagram

Each association between an actor and an use case element in the model has a `TDprob` parameter. Table II presents the distribution among the actors and the use cases modeled into UC shown in Figure 2. For instance, the "Employee" actor has 10% distribution to execute the "Visualize Profile" use case, which represents that ninety five Vusers will execute it.

TABLE II. Percentage distribution among actors and their use cases

Actors	Use Cases	Percentage	VUsers
Headhunter	Visualize Profile	100%	50
			50
Employee	Skills Management	60%	570
	Certifications Management	20%	190
	Experiences Management	10%	95
	Visualize Profile	10%	95
			950
		Total	**1000**

2) Activity Diagram Design: Each use case element is decomposed into a specific AD, in which the use case element and the AD must have the same name. In the example presented in Figure 2 the use case "Skills Management" was decomposed into the AD presented in Figure 3. Basically, an AD represents a sequence of activities performed by an actor over the SUT. In our example, we have modeled an AD using six partition elements, *i.e.* `Login`, `Skill Decision`, `Add Skill by Tree View`, `Add Skill by Finding`, `Save Skills` and `Logout`. Each of these partition elements represents a transaction that is used as a performance metric to measure the response time. For example, the `Login` transaction has two activities, *i.e.* `Home Page` and `Login`. The former represents the user accessing the Skills application home page; and the latter corresponds to a user signing up in the application. In the second transaction, `Skill Decision`, the user selects

SkillsPage to get the list of all registered skills. Then, the user can choose one of the following transactions: Add Skill by Tree View or Add Skill by Finding. The former has one activity: Load Skills; while the latter is formed by Find Skill. Thus, both activities point out to Add Skill and Save Skill in the Save Skill transaction. It is important to highlight that the user fills some information of skills during the Save Skill activity such as proficiency level, acquired date, or last year that the skill was used. The last activity is Logout, which corresponds to the user logging out of the application. It is also important to note that the UML diagrams could be annotated with a set of tagged values representing the performance test information. Thus, in the AD depicted in Figure 3 we have included seven different tagged values and their respective parameters. Therefore, to make it clearer, we inserted a note element to show some tagged values annotated in the control flow element. In this example, the tags TDaction, TDmethod and TDparameters and their corresponding parameters are showed. Each of these parameters has a value that is bounded to the transition between the Add Skill and Save Skill activities.

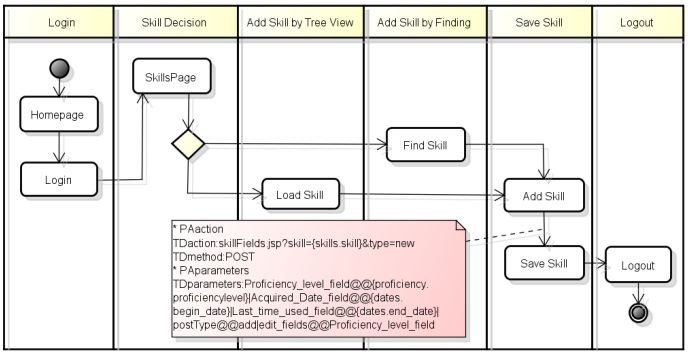

Fig. 3. Skills Management Activity Diagram

B. Abstract Test Cases and Scenarios Generation

After the performance engineer modeled and exported the Skills models to an XML format, this file must be loaded using the PLeTsPerf. Thereunto, the performance engineer click on Import XMI button (see Figure 4) and then selects the desired XMI file. After that, the tool validator module checks if the models are consistent and meet the modeling guidelines of our approach. In case of the models has been validated, the Generate ATC button will be enabled. Thus, the performance engineer must click on that button to generate abstract test cases and scenarios. Figure 5 shows an example representing one of the abstract test cases generated from the AD presented in Figure 3. This abstract test case represents user activities (*e.g.* 1. Homepage and 2. Login) and its related tagged values are presented between double angle quotation marks (*e.g.*, ≪TDmethod: POST≫). Moreover, each transaction is represented by its corresponding partition element name in the AD, (*e.g.* Login and Skill Decision).

C. Test Data Parameters

After generating the abstract test cases and scenarios, it is necessary to perform the parameterization (see Figures 4 and

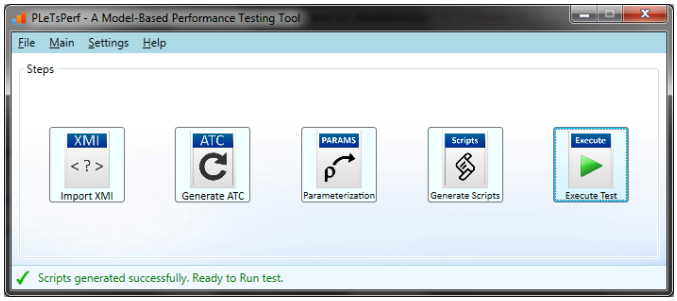

Fig. 4. PLeTsPerf MBT Tool

```
#Abstract Test Case: Skill Management
A Login
 1. Homepage
  <<TDmethod: POST>>
  <<TDaction: mainIE.jsp>>
 2. Login...
B Skill Decision
 3. Skills Page...
C Add Skill by Tree View
 4. Load Skills...
D Save Skill
 5. Add Skill...
 6. Save Skill
  <<TDmethod: POST>>
  <<TDaction: skillFields.jsp?skill={skills.skill}&type=new>>
  <<TDreferer: skillFields.jsp?skill={skills.skill}&type=new>>
  <<TDparameters:
    Proficiency_level_field@@{proficiency.proficiencylevel}|
    Acquired_Date_field@@{dates.begin_date}|
    Last_time_used_field@@{dates.end_date}|
    postType@@add|edit_fields@@Proficiency_level_field>>
E Logout
  7. Logout
```

Fig. 5. Snippet of abstract test case generated from the Skill Management use case

6) of the data parameters, which are designed as dynamic when creating the test models. For instance, in Figure 3 the transition between the Add Skill and Save Skill activities has a set of parameters and each one has an associated value (they are also represented in the abstract test cases and scenarios - see Save Skill activity in Figure 5). In this case, the Proficiency_level_field field requires a dynamic data that is stored in the proficiency file, which contains the proficiencylevel. Therefore, a parameter value is modeled to represent a concatenation of two pieces of information: file name and column name, which are delimited by the squiggly brackets "{" "}", *e.g.* "fileName.ColumnName". The first line in the file represents the header, which must be the column data name. The next lines in this file represent the current data values. It is important to note that the a single file (proficiency) can contain one or more parameters and each one can refer to different activities. Furthermore, both headers and data must be separated by a comma ","- a standard adopted by Comma-Separated Values (CSV) files. Thus, a single CSV file could be used to store all or some tagged values of the Skills models. For example, the TDparameters and TDhost can be stored in a single file. It is worthwhile to mention that one or more parameters can be associated to a single control flow element present in an UC or AD diagrams. For example, in the AD depicted in Figure 3 a parameter could be set into any control flow element, *e.g.*, TDparameters. Hence, when a control flow element is associated to an external file for correlation, the PLeTsPerf requires the execution of the

`Parameterization` step (see Figure 4).

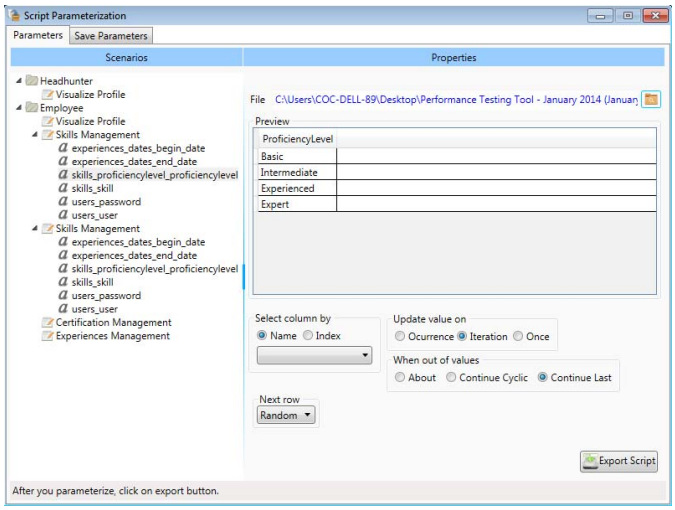

Fig. 6. Test Data Parameterization

D. Test Scripts & Scenarios Generation

The next step to be performed by PLeTsPerf is to instantiate scripts and scenarios to LR from the abstract test cases and scenarios previously generated.

Basically, the LR structures its test scenarios and scripts into two files. One of them is a scenario file that is used to store the test configuration, workload profile distribution among test scripts and the performance counters that will be monitored by the tool. The other one is a script configuration file that stores all information about users' interaction with the application. Figure 7 shows the transaction `Add Skill by Tree View` from a LR script generated from the AD depicted in Figure 3. It is important to note that during the execution of this scripts by each Vuser, the tool will select a data from the file, using a predefined algorithm, to define a value to every parameterized parameters. PLeTsPerf tool has support to three algorithms to select test data from a file: Once, Iteration and Occurrence. For example, depending on the selected algorithm, a data from the CSV file could be used just once by each Vuser for each iteration (option `Iteration` on Figure 6).

E. Test Execution

The tool last step is the test execution. In this step PLeTsPerf invokes the LR, through an internal call, and load the scripts and scenarios previously generated. After that, the test execution is automatically started. After now, the performance engineer must interact directly with the LR interface, performing actions to control and analyse the test execution, *e.g.* stop the test or add a new metric to be monitored. It is important to note that those activities related to the test execution, monitoring and analysis are performed directly by the LR.

V. Lessons Learned

Basically, the main lessons we have learned from the development and the use of our model-based performance testing tool PLeTsPerf are as follows:

```
Action()
{
  ...
  lr_start_transaction("Add+Skill+by+Tree+View");
    lr_start_sub_transaction("Save Skill", "Add+Skill+by+
        Tree+View");
    web_submit_data("Save Skill",
    "Action=skillFields.jsp?skill={skills_skill}&type=new",
    "Method=POST",
    "RecContentType=text/html",
    "Referer=skillFields.jsp?skill={skills_skill}&type=new",
    "Mode=HTTP",
    ITEMDATA,
      "Name=Proficiency_level_field",
      "Value={proficiencylevel_proficiencylevel}", ENDITEM,
      "Name=Acquired_Date_field", "Value={dates_begin_date}"
          , ENDITEM,
      "Name=Last_time_used_field", "Value={dates_end_date}",
          ENDITEM,
      "Name=postType", "Value=add", ENDITEM,
      "Name=edit_fields", "Value=Proficiency_level_field",
          ENDITEM,
    LAST);
    lr_end_sub_transaction("Save Skill", LR_AUTO);
  lr_end_transaction("Add+Skill+by+Tree+View",LR_AUTO);
  ...
}
```

Fig. 7. Snippet of performance test script generated for the LoadRunner

More modeling less recording: An advantage of our MBT approach is that it allows to correct failures in an easier and simpler manner when comparing to CR approach. For instance, during the script design, a performance engineer could make some typing mistakes. However, it is possible to correct them in a fast manner when using our MBT approach, since the performance engineer has the possibility to return to the previous actions in the AD and perform some adjustments, *e.g.*, add an alternative flow, remove actions, or add transactions. Nevertheless, when using performance tools based on CR approach, it is necessary to restart the recording, since these tools do not allow to return to the action previously recorded. It is important to mention that the MBT approach also has a drawback, *e.g.*, the process of annotating performance information through adding parameters into the model is time consuming, and the interface of UML modeling tools is not so intuitive;

Dynamic test data parameterization: In previous versions, our MBT tool, could perform the test data parameterization only in a random manner. However, it is possible to exist a parameter dependent from another parameter previously generated. Then, the next parameter must be associated to the same register. For instance, considering a login script containing two fields in a form: user and password. The load generator randomly selects a data related to the user field, after that, a data related to the password field must be selected based on the same user previously randomized. In the current version this drawback was handled. Thus, using PLeTsPerf, a performance engineering can choose among three ways to parameterize the data selection, *e.g.*, for each interaction, for each occurrence, and just once. Another concern about parameterization is related to possibility of selecting data by reading them from the database, instead of CSV file. Nowadays, most load generators have integration with several SGBDs;

Reverse engineering: During the development of our tool and approach, we could identify evidences of new features based on demands of previous impressions by subjects during the experiment conducted to measure and evaluate both MBT and CR approaches. An important feature is the capability to generate the model based on scripts, *i.e.*, the reverse engineering of our MBT approach;

Lack of a standard performance language/script: One of the issues that motivated our research is the learning curve to replace or upgrade a load generator technology. The previous experiment's results [14] point that reuse of this drawback could be handled if the load generators use the same script or technique. For instance, an alternative to avoid this issue could be use a graphical notation, such as a Domain-Specific Languages (DSL) [16], for modeling performance testing and the load generators have a feature to transform (model transformation) the notation into scripts and scenarios;

Supporting other technology protocols: Although our tool could generate for test cases for Web Applications, a real environment requires several protocols. For this reason, the effort to extend the tool to support other protocols is necessary. Actually, this is a limitation of PLeTsPerf, since it generates performance test scripts and scenarios just for applications based on HTTP request. However, an important standard architecture presents in most of complex applications are related to Web Services consuming by the SUT, which requires the SOAP protocol. In addition, there are other protocols supported by novel applications, *e.g.*, Siebel, Oracle, .NET and Ajax.

VI. Conclusions

In this work we presented how we developed, in collaboration with an IT company, a model-based performance tool, *i.e.* PLeTsPerf. Our tool was developed as part of a process to apply model-based performance testing. The goal of this process is to support all phases of an MBT approach, from the modeling to the test execution. Thus, our process starts with those activities related to the design and construction of test models, which are performed by a performance engineer, and then it describes those activities performed using the PLeTsPerf. Basically, the tool supports the generation of abstract test cases and scenarios, similar to natural language. After, these test cases are instantiated to scripts and scenarios and then are executed by the LR. In order to demonstrate that our tools is suitable to support the adoption of an MBT approach, we presented an example of use conducted in collaboration of our partner company. In this example of use we show how we applied our approach and the tool to generate scripts and scenarios to test a Web application.

It is important to highlight that our partner company started to evaluate the tool in a real test project. In case of the PLeTsPerf meet their requirements, it will be adopted and the tool evolution will be also made by the company. Despite the adoption, our research group will remain involved with the development of the tool, *e.g.* extend it to support more protocols and modeling notations.

Acknowledgment

We thank CNPq/Brazil, CAPES/Brazil, PROCAD, INCT-SEC, and Dell for the support in the development of this work. This study was developed by the Research Group of the PDTI 001/2015, financed by Dell Computers with resources of Law 8.248/91. The authors would also like to acknowledge Cristiano Martins, Edemar Mezacasa Junior, Édio da Silva and Marina de Barros for support the development of PLeTsPerf.

References

[1] G. J. Myers and C. Sandler, *The Art of Software Testing*. New York, NY, USA: Wiley, 2004.

[2] I. Sommerville, *Software Engineering*. Pearson/Addison–Wesley, 2011.

[3] Hewlett Packard HP, "Software HP LoadRunner," Available in: http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/, 2015.

[4] TestOptimal, "TestOptimal Model-based Test Automation," Available in: http://www.testoptimal.com/, 2015.

[5] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[6] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea, *Performance Testing Guidance for Web Applications: Patterns & Practices*. Microsoft Press, 2007.

[7] D. Krishnamurthy, M. Shams, and B. H. Far, "A Model-Based Performance Testing Toolset for Web Applications," *Engineering Letters*, vol. 18, no. 2, pp. 92–106, may 2010.

[8] I. K. El-Far and J. A. Whittaker, *Model-based Software Testing*. John Wiley & Sons, Inc., 2001.

[9] G. Ruffo, R. Schifanella, M. Sereno, and R. Politi, "WALTy: A user behavior tailored tool for evaluating web application performance," in *Proceedings of the Network Computing and Applications, Third IEEE International Symposium*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 77–86.

[10] M. Shams, D. Krishnamurthy, and B. Far, "A model-based approach for testing the performance of web applications," in *Proceedings of the 3rd international workshop on Software quality assurance*. New York, NY, USA: ACM, 2006, pp. 54–61.

[11] L. T. Costa, R. Czekster, F. M. Oliveira, E. M. Rodrigues, M. B. Silveira, and A. F. Zorzo, "Generating Performance Test Scripts and Scenarios Based on Abstract Intermediate Models." in *Proceedings of the 24rd International Conference on Software Engineering and Knowledge Engineering*. San Francisco, CA, USA: Knowledge Systems Institute Graduate School, 2012, pp. 112–117.

[12] M. B. Silveira, E. M. Rodrigues, A. F. Zorzo, H. Vieira, and F. Oliveira, "Model-Based Automatic Generation of Performance Test Scripts," in *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*. Miami, Florida, USA: Knowledge Systems Institute Graduate School, 2011, pp. 258–263.

[13] E. M. Rodrigues, F. M. de Oliveira, L. T. Costa, M. Bernardino, A. F. Zorzo, S. d. R. S. Souza, and R. Saad, "An empirical comparison of model-based and capture and replay approaches for performance testing," *Empirical Software Engineering*, pp. 1–30, 2014.

[14] E. M. Rodrigues, F. M. Oliveira, M. Bernardino, R. S. Saad, L. T. Costa, and A. F. Zorzo, "Evaluating Capture and Replay and Model-based Performance Testing Tools: An Empirical Comparison," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: ACM, 2014, pp. 9:1–9:8.

[15] OMG, "UML Profile for Schedulability, Performance, and Time Specification - OMG Adopted Specification Version 1.1, formal/05-01-02," Available in: <http://www.omg.org/spec/SPTP/1.1/PDF>, 2015.

[16] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.