ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/216763444

Generation of Scripts for Performance Testing Based on UML Models

Conference Paper · January 2011

CITATIONS
20

READS
540

6 authors, including:

Maicon Bernardino Da Silveira
Universidade Federal do Pampa (Unipampa)
56 PUBLICATIONS **159** CITATIONS
SEE PROFILE

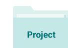
Elder Rodrigues
Universidade Federal do Pampa
60 PUBLICATIONS **157** CITATIONS
SEE PROFILE

Avelino F. Zorzo
Pontifícia Universidade Católica do Rio Grande do Sul
138 PUBLICATIONS **1,212** CITATIONS
SEE PROFILE

Leandro T. Costa
Pontifícia Universidade Católica do Rio Grande do Sul
7 PUBLICATIONS **62** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Referenciais de Educação em Computação View project


Detecting Encrypted Attacks View project

All content following this page was uploaded by Maicon Bernardino Da Silveira on 06 June 2014.

The user has requested enhancement of the downloaded file.

Generation of Scripts for Performance Testing Based on UML Models

Maicon B. da Silveira, Elder M. Rodrigues, Avelino F. Zorzo,
Leandro T. Costa, Hugo V. Vieira and Flávio M. de Oliveira
Faculty of Informatics (FACIN) – Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre – RS, Brazil
bernardino@acm.org, elder.rodrigues@acad.pucrs.br, avelino.zorzo@pucrs.br,
leandro.teodoro@acad.pucrs.br, hugovares@gmail.com and flavio.oliveira@pucrs.br

***Abstract*—Software testing process has a high cost when compared to the other stages of software development. Automation of software testing through reuse of software artifacts (e.g. models) is a good alternative for mitigating these costs and making the process much more efficient and efficacious. Model-Based Testing (MBT) is a technique to automatic generation of testing artifacts based on software models. For software development, the most spread modeling language in either the industrial or academic environments is UML. In such environments, it is desirable to reuse UML models also for MBT, avoiding re-building a different model exclusively for testing automation. These are the main reasons that make these semi-formal models an alternative to implementing MBT. Even though there are a lot of testing tools available commercially, to the best of our knowledge, none of them fully uses MBT. Therefore, this paper describes a case study showing how to implement the MBT process to automate test scripts generation and execution in a real-world, context. Furthermore, our solution is generated automatically by a Software Product Line (SPL).** [1]

***Keywords* - Model-Based Testing; Software Product Line; Performance Testing.**

I. Introduction

Currently, a great number of people and companies use computer programs to automate their activities, delegating to systems the execution of complex tasks. This widespread use of computer systems has also increased the number of residual software or hardware faults that generate failures to users [1] [2]. Therefore, it is important that during the development of a system, different techniques are applied to guarantee that the system provides a service that can be trusted. The ability to deliver a service that can justifiably be trusted is known as dependability [1]. The main attributes that integrate the dependability are reliability, availability, security, confidentiality, integrity and maintainability. According to the taxonomy presented in [1], system dependability can be achieved by four techniques: 1) Fault Prevention - *prevent the occurrence or introduction of faults*; 2) Fault Tolerance - *avoid service failures in the presence of faults*; 3) Fault Removal - *reduce the number and severity of faults*, and; 4) Fault Forecasting - *to estimate the present number, the future incidence, and the likely consequences of faults*. Several works that provide system dependability through fault tolerance, fault prevention and fault forecasting are present in the literature [3] [4] [5].

Although all techniques are used to achieve software dependability, the most used technique in all areas of software development in industry is fault removal, through software testing. Software testing is a process that focus on finding program failures[2] during runtime [6], or that has activities to validate the requirements of a program, determining if the expected results are met [7]. Performance is a key component of reliability and availability; therefore, performance testing is a major activity in system fault removal. However, due to the systems evolution and their amount of features, systems are becoming so complex that testing them is a difficult task. Therefore, it is necessary to implement a testing process to mitigate testing execution on the final product. This process should aim to reduce the costs impact, improving the quality of the software product [2].

One of the techniques that improves the software testing process is Model-Based Testing (MBT) [8]. This technique consists in the generation of test cases and/or test scripts based on the application model. Besides, it also includes the specification of the features that will be tested [9].

In previous work [10] [11], we have used MBT to build testing tools for security and functional testing. These tools were derived products from a Software Product Line [12] called PLeTs [11]. The work presented in this paper expands our previous work to apply MBT in the generation of a new product to execute performance testing. This new product generates test scripts for a commercial tool called LoadRunner [13]. Basically, we include stereotypes in the system UML models to express performance information that will be used during the execution of test scripts.

This paper is organized as follows. Section II presents a short description of MBT, SPL and the PLeTs architecture. In Section III we show how we have used stereotypes to include information on the UML models and a brief description of the LoadRunner tool. In Section IV we apply our strategy to an actual case study that uses the LoadRunner tool to execute performance testing. This case study is a tool used in a major

[1]Study developed by the Research Group of the PDTI 001/2011, financed by Dell Computers of Brazil Ltd. with resources of Law 8.248/91.

[2]We use fault, error, and failure definition from [1].

IT company. Finally, Section V summarizes the contributions of our work.

II. Background

Software modeling is an important technique that is used in software development because it allows to capture and to share knowledge about a system. During the development process, information about the system is described in many different documents. To include this information, through, for example, UML stereotypes, enriches the specification documents. This increases the quality of the specification and the use of models developed as the system evolves [14]. Thus, this information is used to model the incremental creation of new artifacts, or even allows the automation of other processes to improve the quality of different developed artifacts. This section describes two approaches to automate software artifacts development based on features that are included in the system model: Model-Based Testing and Software Product Line.

A. Model-Based Testing

Usually, system behavior and requirements are described using a formal or semi-formal model, allowing team members to share and to use these models during the life-cycle of the software development. In spite of that, test engineers are still producing test cases or test scripts in an informal way based on a mental representation that they create. A better alternative would be to derive test artifacts from the system models [15]. This strategy is commonly known as Model-Based Testing (MBT) [8]. MBT is based on the idea of the automation of test case/scripts generation. Albeit, the use of existing models can increase productivity during the process of software testing [14].

The cost of software testing is related to the number of interactions and test cases that are executed during the development process. As it is one of the most costly and expensive phases of software development [6], MBT is a good approach to mitigate this problem by automating the process of generating test cases or scripts [16].

Several works on MBT have been produced in the past years. A systematic review is presented in [17] to evaluate quantitatively and qualitatively some features of MBT. In this study, 78 articles were evaluated and the following items were reviewed: type of model (formal or semi-formal) used for test generation; test types in which the approach can be applied to; level of automation; support tools; criteria for test coverage, etc. The study presents test domains (system, integration, unit/component, regression) in which each of the works were applied to. The survey identifies that most of the works use MBT in system testing domain (66%). System testing includes Performance testing, which is the focus of our work.

Some of the works mentioned in [17] use the same MBT process, for example [8] [18]. This process requires specific activities in addition to the usual activities of software testing. This will require that the test engineers adjust their testing process, and invest in the use and training of new tools. The main activities that define the MBT process are (see

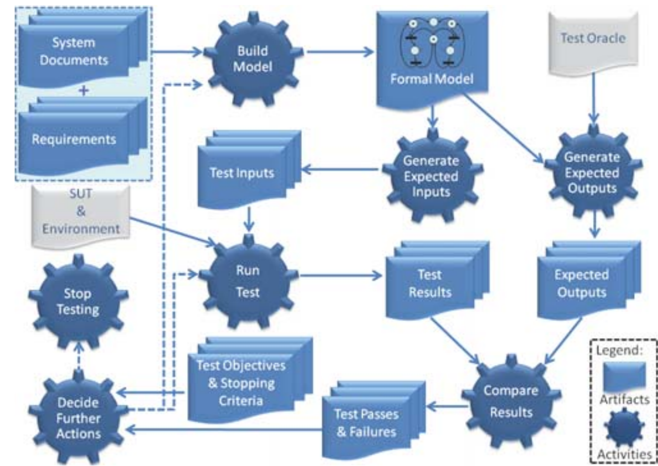


Fig. 1. Activities for MBT [8]

Figure 1) [8]: *Build Model*, *Generate Expected Inputs*, *Generate Expected Outputs*, *Run Tests*, *Compare Results*, *Decide Further Actions* and *Stop Testing*. 1) *Build Model*: constructs a model based on the specification of system requirements. This step defines the choice of the model, according to the application being developed; 2) *Generate Expected Inputs*: uses the developed model to generate test inputs (test cases, test scripts, application input); 3) *Generate Expected Outputs*: generates some mechanism that determines whether the results of a test execution are correct or not. This mechanism is the test oracle and it is used to determine the correctness of the output; 4) *Run Tests*: executes test scripts and stores the processing results of each test case. This execution can be performed in the system under test (SUT) and/or system's environment; 5) *Compare Results*: compares the test results with expected outputs (test oracle), generating reports to alert the test team about failures; 6) *Decide Further Actions*: based on the results, it is possible to estimate the software quality. Depending on the quality achieved, it is possible to stop testing (quality achieved), to modify the model to include further information to generate new inputs/outputs, to modify the system under test (to remove remaining faults), or to run more tests; 7) *Stop Testing*: concludes the system testing. The activities from MBT process can bring several new advantages to the test team [8], for example: shorter schedules, lower cost, and better quality.

A good possibility to reduce the problems mentioned at the beginning of the previous paragraph would be to have a single tool that would cover all the phases of the MBT process, i.e. a tool in which it would be possible to describe the system model, that would generate test cases/scripts, that would execute the test scripts and also compare the results. Even better if the test team could have a tool that could generate the testing tool for each different application or different type of test the test team wants to execute over the same application. Furthermore, it is desirable that the test team reuse implemented artifacts (e.g.: models, software components, scripts). Thus, in this context, it becomes interesting

to design a set of MBT tools based on a Software Product Line (SPL). A SPL ensures the variability, reusability of test artifacts, thus decreasing costs and time to market. Several evidence of the benefits of the use of SPL, in different areas, can be found in [19] [20]. The next section introduces the concept of SPL.

B. Software Product Line

A Software Product Line (SPL) seeks to exploit the commonalities among systems from a given domain, and at the same time to manage the variability among them [12]. According to [21], SPL engineering has three main concepts: core assets development, product development, and management. The core assets are the main part of an SPL, and its components aim to represent, in a clear way, the common and variable aspects of the future products. Thus, following the SPL concepts, new products variants can be quickly created based on a common architecture, models, software components, etc. Because of that, SPL allows for rapid entry of a product on the market as well as makes it easier for mass customization of products of a company. Companies are finding that the practice of building sets of related systems from common assets can, in fact, produce quantitative improvements in product quality and consumer satisfaction, efficiently meeting the current demand for mass customization.

However, given the large number of products that can be present in a product line (PL), it is necessary to control the variability and commonalities among them. The variability management is used to control the variables aspects present in the products of the PL.

Feature Models is an important concept to modeling variability. Originally, Feature Modeling was developed as part of Feature-Oriented Domain Analysis (FODA) [22]. However, nowadays it is applied in many areas such as embedded systems [23] or networks protocols [24].

When Feature Models are applied to represent variability, they are analyzed and categorized as common, optional or alternative [25]. *Common features* represent features that must be present in every product of the SPL. There are also called mandatory, necessary, or kernel features. *Optional features* represent features that are supported by some products in the SPL, and; the *alternative features* represent features that are mutually exclusive, i.e. only one of the features can be provided in each product of the SPL (see Figure 2).

C. PLeTs Tool

The PLeTs tool [11] aims to automate the generation, execution and results collection of MBT process. The tool is able to manage the whole MBT process and is based on the concepts of SPL. Its goal is not only the reuse of artifacts to make it easier and faster to develop a new tool of the family, but also to improve the creation, run and gathering of test results. It was developed with the intent to be used by software engineers, developers and test engineers, assisting the process of defining and executing test cases and test scripts. Figure 2 shows the current PLeTs Feature Model that represents some of the features that could be present in a software variant. The first level of the model has four main features:

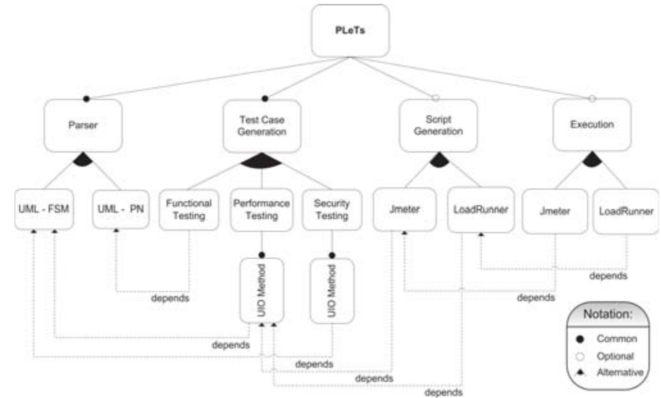


Fig. 2. Feature Model for PLeTs Tool [11]

1) *Parser*: represents the *Build Model* step in the MBT main activities (see Section II-A). It is a mandatory feature and has two child features: UML - FSM and UML - PN. Each one of these parsers is used to extract the information from the UML models to generate a formal model (Finite State Machine (FSM) or Petri Nets (PN)); 2) *Test Case Generation*: represents part of the *Generate Expected Inputs* step in the MBT main activities. It is a mandatory feature and has three child features: Functional Testing, Performance Testing and Security Testing (one of them should be selected in each software variant). Both Performance Testing and Security Testing have a mandatory child feature: UIO Method [26]; 3) *Script Generation*: represents another part of the *Generate Expected Inputs* step in the MBT main activities. It is an optional feature, because, for example, for security testing there could be no tool to execute the generated test cases. This feature has two child features: Jmeter [27] and LoadRunner [13]; 4) *Execution*: represent the *Run Tests and Compare Results* step in the MBT main activities. This feature also has two child features: Jmeter and LoadRunner.

It is important to highlight that there are dependencies between some features (see the dotted lines in Figure 2). For example, if some software variant selects the feature *Execution* and the child feature *LoadRunner*, it must select the feature *Script Generation* and the child feature *LoadRunner*, because the tool is not able to execute the tests without a test script.

Another important point is that the Feature Model can be extended to support new testing techniques or tools, adding new child features to the main four features. For example, if someone wants to add new features to work with the SilkPerformer tool [28], he should include new child features for the *Script Generation* and *Execution* main features.

To develop the tool in a way to represent the feature model flexibility, the PLeTs architecture is based on plug-ins that allows extensibility and flexibility. Based on that, the tool allows to select, in runtime, each plug-in (represented by a feature) that is necessary to perform a MBT activity and to automatically generate/execute the test scripts.

III. UML Models for Performance Testing

In previous works [10] [11], we described some components of PLeTs developed for security and functional testing. Here we apply our approach in a different application domain, reusing, or expanding, previous components automatically through SPL.

In our approach, the starting point for test script generation is the construction of an UML model activity diagram with performance stereotypes[3], represented in an XMI file. This XMI file is parsed and converted into a formal model, e.g. Finite State Machine (FSM). Then, performing the UIO Method [26], the sequences of activities that have to be executed are obtained. These sequences could be transformed in a description that is equivalent to test cases in natural language. This approach was used in our previous works, i.e. all these features have already been developed. To expand our work, we propose here a set of new features to support the generation of a product that can perform performance testing for the LoadRunner tool [13].

As mentioned above, we use stereotypes, which are the way we describe performance information necessary to generate our test cases and test scripts. We include stereotypes in two UML diagrams: use case and activity. Our approach uses four stereotypes from our previous works and a new one that was missing. The five performance stereotypes are the following: 1) `<<PApopulation>>`: this stereotype has two tags: the first one represents the number of users that are running the application, while the second one represents the host where the application is executed (defined in all actors of the use cases diagram); 2) `<<PAprob>>`: defines the probability of execution for each existing activity; 3) `<<PAtime>>`: expected time to perform a given use case; 4) `<<PAthinkTime>>`: denotes the time between the moment the activity becomes available to the user and the moment the user decides to execute it, for example, the time for filling a form before its submission; 5) `<<PAparameters>>`: defines the tags for the input data that will be provided to the application when running the test scripts (this is a new stereotype that our previous works did not include).

A. LoadRunner

HP LoadRunner [13] is a product to analyze the behavior and the performance of systems. This tool can emulate hundreds, or thousands, of users, known as Virtual Users (VUsers), simultaneously.

In the LoadRunner architecture the main configuration part that has to be changed for each application that is tested is stored in the script folder. Basically, our PLeTs plug-in generates a new *script* file, that are scripts written in C language. The test description, which includes application transactions and parameters, is included in that file. Our PLeTs plug-in also generates the configuration scenarios, which are included in *scenarios* file. This file contains performance counters that will be used in the reports that are generated by the LoadRunner.

[3]We use UML 2.0 SPT (Schedulability, Performance and Time) Profile [29] [30].

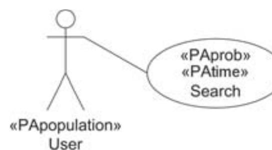


Fig. 3. Skill Management Use Case Diagram

Another important LoadRunner feature is a library with predefined functions. Each set of functions that are included in this library have a specific task in each of the testing protocols that LoadRunner implements. For example, functions starting with `web` are used to represent HTTP requests, while the ones starting with `lr` are general and can be used in all protocols. Some of the existing functions, which will be used in the work described in this paper, are: 1) lr_think_time: determines the idle time between user interactions and the system; 2) web_submit_data: submits web form data without a previous operation context; 3) web_url: is responsible for accessing a URL via a web browser; 4) web_image: represents a click on an image on a page (tag HTML <IMG>); 5) web_link: represents a click on a text link on a page (tag HTML <A href=...> e </A>); 6) web_submit_form: submits web form data but considering the context of the previous operation.

Based on these functions, it is possible to simulate the test scenarios in order to verify the performance behavior of a given application. Moreover, the concepts discussed in this section (tool architecture, script features and functionalities interpreted by LoadRunner), were important to implement the automatic scripts generation based on information extracted from the UML model.

IV. Case Study: Skills Management Tool

In this section, we apply our strategy and the PLeTs tool to an application that manage skills, certifications and experience of employees of a given organization. This tool is called Skills and was developed in collaboration between a research group of our institution and an IT company. Skills was developed in the Java programming language, using the MySQL database for data persistence and Tomcat as web application server.

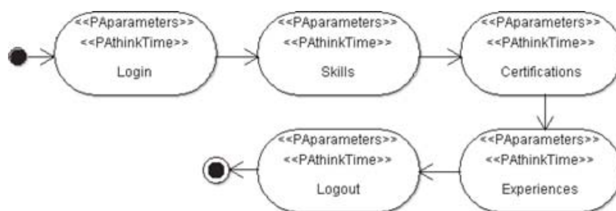


Fig. 4. Skill Management Activity Diagram

One example of our use of UML with stereotypes is the “Search” case. Figure 3 shows part of the user interaction behavior with the application. Furthermore, the necessary steps

to implement this case study are detailed in the activity diagram shown in Figure 4. This diagram represents five sequential activities, starting with “Login” to access the system, “Skills” to consult the user’s abilities, “Certifications” to view the technical certification assigned to the user; “Experiences” a list the user’s professional experience; and “Logout” to exit the system.

Once all the UML diagrams (e.g. see Figures 3 and 4) have been constructed, we use PL*e*T*s* to generate the test scripts for the Skill Management Application. The scripts were generated to run on LoadRunner, but we could have changed the plug-in that generates scripts and apply the same set of test using a different testing tool, for example the IBM Rational Performance Tester [31]. Figure 5 shows an extract from the script that was generated. This extract shows the actions of a given user, for example the think time through function `lr_think_time(10)` and the submission of data filled in the step “Login” (from the activity diagram - see Figure 4) through function `web_submit_form`.

```
Action()
{
web_url("Search",
"URL=http://192.168.1.26/skillsApp/mainIE.jsp",
"Resource=0",
"RecContentType=text/html",
"Referer=",
"Mode=HTML",
LAST);

lr_think_time(10);
web_submit_form("Loging.jsp",
ITEMDATA,
"Name=username", "Value=admin", ENDITEM,
"Name=password", "Value=123456", ENDITEM,
LAST);
```

Fig. 5. Script Generated for LoadRunner

As described in Section III, we can include five stereotypes in our UML diagrams, with one or more tags. As can be seen in Figure 3, the “Search” use case diagram has three of those stereotypes, and they are generated with the following values:

- $PApopulation$
  - $TDpath$ = http://192.168.1.26/skillsApp/mainIE.jsp
  - $TDpopulation$ = 30
- $PAprob$
  - $TDprob$ = 1.0
- $PAtime$
  - $TDtime$ = 300

The other stereotypes, in this case study, are included in the activity diagram as shown in Figure 4. For example, the “Login” activity has the following values:

- $PAthinkTime$
  - $TDthinkTime$ = 00:00:10
- $PAparameters$
  - $TDaction$ = login.jsp
  - $TDparameters$ = username@@admin
  - $TDparameters$ = password@@123456

Notice that the `TDparameters` tag is the concatenation of two pieces of information: name and value, separated by the delimiter @@. This information could be generated automatically for different scenarios that the test engineer wants to test.

All tags are extracted from the UML diagram and processed by the PL*e*T*s* plug-in that generates scripts for LoadRunner. Although all necessary information is included in the UML diagrams, LoadRunner also needs a description for the test scenarios. The configuration of the test scenarios is included in the *scenarios* file. The PL*e*T*s* plug-in uses a template to automatically generate the *scenarios* file for LoadRunner. This template has some markings that are replace by the tags from the UML diagrams. For example, tag `<<Vusers>>` represents the number of virtual users that will be simulated during the test.

Besides the `<<Vusers>>` marking, the template file for scenarios has other markings that are used when generating test scripts: 1) `<<Path>>`: contains the application host address; 2) `<<Result_file>>`: specifies the results file name; 3) `<<HostGenerator>>`: defines the server that generates the load; 4) `<<TestChief>>`: contains the paths for the script tests that will be run for the scenario; 5) `<<GroupChief>>`: stores information on each VUsers group that will be simulated by the test script; 6) `<<GroupInfo>>`: defines the performance counters that will be used during the test.

Once the test script generation is completed, PL*e*T*s* calls the LoadRunner tool passing as a parameter the test script and scenario produced. In the test we have performed, we used the LoadRunner standard performance counters that are already set in the default user interface. We could have redefined the screens and counters we wanted to verify, but for the Skill Management application this was not necessary.

V. Conclusion

The use of formal models is a good way for modeling the behavior and structure of a system. This allows for a precise understanding of the system behaviour by software developers or test engineers and, therefore, allows for a better reuse of the system components as the system evolves.

Although automatization of software testing is desirable, test engineers usually perform testing manually using, of course, a set of tools. Capture-replay tools, for example, are widely used, but they require a full, running build of the application in order to create the test scripts for the first time, thus delaying script development; moreover, they require the tester to execute (manually!) all the script al least once. Besides, test engineers have to build a mental model of the whole testing process or strategy. As shown in this paper, MBT can help test engineers to build a formal model for their testing process as well. It is important to mention that despite all the advantages of using MBT described in this paper, MBT requires a professional with skills in programming languages, software testing and modeling, and also with theoretical background

on mathematics, automata theory, graph theory and formal languages [8].

Furthermore, the use of MBT could require a significant investment if not supported by a set of tools to plan, monitor and produce test artifacts. This paper has shown how we have applied MBT to an application without having to start the whole process from scratch, hence helping test engineers that do not have the above mentioned skills or background. Our approach is based on an SPL tool that generates testing tools based on MBT. In some previous works we had already applied our strategy to different testing domains and we were able to reuse some already developed testing artifacts in the work presented here. This has reduced our costs to test an actual application that is used in an IT company.

Another important contribution of this paper is the development of a tool to allow the use of MBT in commercial tools, such as the LoadRunner. Currently there are none, to the best of our knowledge, commercial tool that uses MBT. The LoadRunner plug-in developed for the PL*e*T*s* tool shows that the use of MBT is viable in industry. Besides, the team of the IT company can now use the same plug-in for different applications they want to test.

Despite the advantages of the implementation of the new plug-in presented here, some improvements in this new plug-in could be adopted, for example, in the current plug-in implementation several data that are needed for the generation of the LoadRunner scripts are inserted in the model stereotypes (See Section IV). A simple modification could be the insertion, in the `TDparameters` tag, of a name of a file, or even a database, that would contain these data. The plug-in, then, would generate scripts based on the data stored in that file, or database. These improvements will allow us to use parts of the plug-in for different commercial tools and to further explore our strategy.

Acknowledgment

We also thank CNPq/Brazil, CAPES/Brazil and INCT-SEC for the support in the development of this work. We also thank the reviewers for their comments, which helped to improve our paper.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transaction Dependable Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[2] M. Young and M. Pezzè, *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[3] J.-C. Laprie, "Dependability Evaluation of Software Systems in Operation," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 701–714, 2009.

[4] D. Powell, "Failure mode assumptions and assumption coverage," *22nd International Symposium on Fault-Tolerant Computing. Digest of Papers.*, pp. 386–395, 2002.

[5] A. Romanovsky and A. F. Zorzo, "Coordinated atomic actions as a technique for implementing distributed gamma computation," *Journal System Architecture*, vol. 45, no. 15, pp. 1357–1374, 1999.

[6] G. J. Myers and C. Sandler, *The Art of Software Testing*. New York: John Wiley & Sons, 2004.

[7] B. Beizer, *Software System Testing and Quality Assurance*. New York: Van Nostrand Reinhold, 1984.

[8] I. K. El-Far and J. A. Whittaker, *Model-based Software Testing*. New York: Wiley, 2001.

[9] M. Popovic and I. Velikic, "A Generic Model-Based Test Case Generator," *12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pp. 221–228, 2005.

[10] K. P. Peralta, A. M. Orozco, A. F. Zorzo, and F. M. Oliveira, "Specifying Security Aspects in UML Models," *1st International Workshop on Modeling Security In ACM/IEEE 11th International Conference on Model-Driven Engineering Languages and Systems*, pp. 1–10, 2008.

[11] E. de M. Rodrigues, L. D. Viccari, A. F. Zorzo, and I. M. Gimenes, "PLeTs-Test Automation using Software Product Lines and Model Based Testing," *22th International Conference on Software Engineering and Knowledge Engineering*, pp. 483–488, jul. 2010.

[12] P. Clements, L. Northrop, and L. M. Northrop, *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing, 2001.

[13] Hewlett Packard - HP, "*Software* HP LoadRunner," Available in: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn= bto&cp=1-11-126-17$\hat{8}$_4000_100, sep. 2010.

[14] L. Apfelbaum and J. Doyle, "Model Based Testing," *Software Quality Week Conference*, 1997.

[15] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, *Model-Based Testing of Reactive Systems: Advanced Lectures*. Secaucus: Springer, 2005.

[16] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer," *Formal Methods and Testing*, 2008.

[17] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-Based Testing Approaches: A Systematic Review," *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pp. 31–36, 2007.

[18] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing," Working Paper, The University of Waikato, Hamilton, New Zealand, Tech. Rep., apr. 2006.

[19] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, "Introducing PLA at Bosch Gasoline Systems: Experiences and Practices," *3rd International Conference on Software Product Lines*, vol. 3154, pp. 34–50, 2004.

[20] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley Longman, 2003.

[21] Software Engineering Institute (SEI), "Software Product Lines (SPL)," Available in: http://www.sei.cmu.edu/productlines/, sep. 2010.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon University, SEI, Tech. Rep., nov. 1990.

[23] K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker, "Generative Programming for Embedded Software: An Industrial Experience Report," *1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pp. 156–172, 2002.

[24] M. Barbeau and F. Bordeleau, "A Protocol Stack Development Tool Using Generative Programming," *1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pp. 93–109, 2002.

[25] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing, 2004.

[26] R. Anido and A. Cavalli, "Guaranteeing Full Fault Coverage for UIO-Based Testing Methods," *8th International Workshop for Protocol Test Systems*, pp. 221–236, 1995.

[27] Y. Jing, Z. Lan, W. Hongyuan, S. Yuqiang, and C. Guizhen, "JMeter-based aging simulation of computing system," *International Conference on Computer, Mechatronics, Control and Electronic Engineering*, vol. 5, pp. 282–285, aug. 2010.

[28] G. hun Kim, H. choun Moon, G.-P. Song, and S.-K. Shin, "Software Performance Testing Scheme Using Virtualization Technology," *4th International Conference on Ubiquitous Information Technologies Applications*, pp. 1–5, dec. 2009.

[29] OMG, "UML Profile for Schedulability, Performance, and Time Specification - OMG Adopted Specification Version 1.1," Available in: http://www.omg.org/spec/SPTP/1.1/, 2005.

[30] C. Woodside and D. Petriu, "Capabilities of the UML Profile for Schedulability Performance and Time (SPT)," in *Workshop SIVOES-SPT RTAS'2004*, 2004.

[31] D. C. et al., *Using Rational Performance Tester Version 7*. IBM Redbooks, 2008.

View publication stats