Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

A Software Product Line for Model-Based Testing Tools

Elder M. Rodrigues, Avelino F. Zorzo, Itana M. Gimenez
Elisa Y. Nakagawa, Flávio M. Oliveira, José C. Maldonado

Technical Report N^o 069

Porto Alegre, December 2012

Resumo

Software testing is recognized as a fundamental activity for assuring software quality. A diversity of testing tools has been developed to support this activity, including tools for Model-Based Testing (MBT). MBT is a testing approach to automatic generation of testing artifacts from the system model. This approach presents several advantages, such as lower cost and less effort. Nevertheless, most of the time, MBT is applied in an *ad hoc* manner. In another perspective, Software Product Line (SPL) offers possibility of systematically generating software products at lower costs, in shorter time, and with higher quality. The main contribution of this paper is to present a SPL for testing tools that support MBT. We have applied our strategy to develop MBT testing tools in the context of an IT company. As the results, we have observed that efforts to build new testing tools was reduced considerably and, therefore, SPL can be considered a very relevant approach to improve productivity and reuse during generation of testing tools.

1 Introduction

The demand for new software systems applied to different areas of knowledge, or even to new types of applications, has increased in the past years. This has led companies to produce software on an almost daily basis, putting a strong pressure on software engineers that have to ensure that the software is working properly. This situation has increased due to the competition pressure as several companies are paying attention to the mistakes made by their rivals. High quality software is, therefore, a very important asset for any software company nowadays. Hence, software engineers are looking for new strategies to produce and verify software in a very fast and reliable manner.

One of the most important activities for assuring quality and reliability of software products is by means of fault[1] removal, *e.g.*, **Software Testing** [Myers, 79] [Harrold, 00]. Software testing is applied to minimize the number and severity of faults in a software. Complementary approaches, such as fault tolerance, fault forecasting, fault prevention, or even fault removal in the sense of software verification, could also be applied to increase software quality [Avizienis *et al.*, 04].

Software testing can contribute to increase software quality; however it is important to systematically undertake the testing activity by using well-defined testing techniques and criteria [DeMillo *et al.*, 78] [Myers, 79] [Rapps, 85]. Despite the variety of testing techniques, criteria and tools that are available, most of them are applied in industry in an *ad hoc* manner, because software engineers still have to develop test scripts, provide test cases, and understand and write the tools configuration files. These activities can be error prone and faults can be injected in one of these tasks of software testing, mainly because those are manual tasks.

In this context, Model-Based Testing (MBT) is one technique that has gained considerable attention to support the testing activity by taking into account information represented in system models [Utting, 06]. MBT has several advantages when compared to other testing techniques, *e.g.*, the reduced probability of misinterpretation of the system requirements by a test engineer. In addition, it helps to reduce the overall testing workload, since it increases the automation and reuse of testing artifacts [Abbors *et al.*, 10] [El-Far, 01]. Furthermore, the availability of MBT testing tools has also made testing a more systematic activity; thus, minimizing cost, effort, as well as reducing the number of remaining faults caused by human intervention. Currently, there is a diversity of commercial, academic, and

[1] We use the definition of fault, error and failure presented in [Avizienis *et al.*, 04].

open source testing tools that automate software testing tasks [Misura *et al.*, 05] [Ma *et al.*, 06]. However, most of these tools have been individually and independently implemented from scratch based on a single architecture. Thus, they face difficulties of integration, evolution, maintenance, and reuse.

In order to reduce these difficulties that MBT faces, it would be interesting to have a strategy to automatically generate specific products, *i.e.*, testing tools, for executing all MBT phases based on the reuse of assets and on a core architecture. This is one of the main ideas behind **Software Product Lines** (SPL). Basically, a SPL is composed of a set of common features and a set of variable parts that represent later design decisions [Clements, 01]. The SPL adoption has increased in the past years and several successful industry cases have been reported in the literature [Bass *et al.*, 98] [Northrop, 02].

Another concept applied to promote reuse and minimize the difficulties in integration, evolution, and maintenance of testing tools is **Reference Architecture** [Garlan, 00]. This concept has emerged to facilitate the reuse of design expertise by achieving solid and well-recognized understanding of a specific domain. There are different initiatives of reference architectures for the Software Engineering domain [Boudier *et al.*, 88] [Nakagawa *et al.*, 11]. Reference architectures for other domains, such as Embedded Software [Saudrais, 11] and Web Browsers [Grosskurth, 05] have also been proposed. In this paper, we focus on a reference architecture for software testing tools (RefTEST) that aims at supporting the development of testing tools [Nakagawa *et al.*, 07]. This architecture has presented a valuable contribution to the development of testing tools [Ferrari *et al.*, 10] [Nakagawa *et al.*, 10]. Nonetheless, SPL and Reference Architecture are two research topics that need to be more investigated in a joint way, in particular, in the domain of software testing.

In this context, this paper proposes to use MBT to systematize software testing. This is achieved by new testing tools to support MBT that are generated by our SPL named PLeTs - **P**roduct **L**ine for Mod**e**l-based **T**esting tool**s** (PLeTs) [Rodrigues *et al.*, 10] . Furthermore, this SPL has a core architecture based on a reference architecture, specifically the RefTEST. PLeTs supports, therefore, the generation of testing tools that automate the MBT process activities, such as generation of test cases, generation of test scripts, and execution of the system under test (SUT) [Silveira *et al.*, 11].

This paper is organized as follows. Section 2 presents background about testing automation, SPL and Reference Architecture. Section 3 presents the proposed SPL for MBT tools. Section 4 describes the architecture and implementation of PLeTs. Section 5 describes how to derive new testing tools using our SPL. Sec-

tion 6 provides some discussion and lesson learned based on the use of PLeTs in an IT company. Finally, conclusions are presented in Section 7.

2 Background

Software testing is a relevant activity to ensure software quality; however, it is expensive, error-prone, and time-consuming [Whittaker, 00]. Thus, the development of testing tools is a constant concern [Bertolino, 07] [Young, 05]. A wide amount of testing tools has been implemented in the past years to automate testing techniques and criteria. They apply diverse types of testing techniques, such as performance testing, stress testing, and security testing [Veanes *et al.*, 08] [Ferrari *et al.*, 10]. Besides that, these tools aim at testing different types of applications, from web systems to critical systems. Therefore, the community of software development has understood that the automation of the testing activity is fundamental. In this context, MBT is one of the approaches to help automate the testing activity, through the automatic generation of test cases or scripts based on information included in the system models [Veanes *et al.*, 08]. Several works on MBT have been produced in the past years [Veanes *et al.*, 08] [Stefanescu, 09] [Abbors *et al.*, 10]; some of them have proposed tools to automate the testing process for different testing techniques and domains, *e.g.*, web application. Moreover, most of the MBT tools use the same process. El-Far [El-Far, 01] presented the main activities that define the MBT process: *Build Model*, *Generate Expected Inputs*, *Generate Expected Outputs*, *Run Tests*, *Compare Results*, *Decide Further Actions* and *Stop Testing*. Even though several tools have been produced, has not been enough investigation on how to leverage and combine the respective common characteristics/activities of these tools derive new testing tools. Moreover, there are a large amount of industrial testing suites that propose to automatize the test case/script creation and execution, *e.g.*, using Capture & Playback [Guckenheimer, 06] [LoadRunner]. There are also some MBT tools, such as Conformiq Qtronic [Huima, 07], that generate scripts to be executed by other testing tool, however these tools are not designed to be integrated with different tools, are limited to functional testing and are independently implemented from scratch based on a single architecture.Thus, it is relevant to investigate how to combine the common characteristics of the MBT tools and the existing testing tools to develop a new MBT testing tool.

One way to explore this idea is to use the concepts applied in Software Product Lines (SPL). A SPL seeks to exploit the commonalities among systems from

a given domain, and at the same time to manage the variability among them [Clements, 01]. According to SEI (Software Engineering Institute) [SEI], SPL Engineering has three main concepts: core assets development, product development and, management of the product line. The core assets are the main part of a SPL, and its components represent, in a clear way, the common and variable aspects of the future products. Furthermore, a SPL could be derived from two main approaches [Krueger, 01]: extractive and reactive. The extractive approach implements a SPL based on a set of single software systems already developed, whereas the reactive approach incrementally increases the SPL when new software is demanded. Thus, following the SPL concepts, new product variants can be quickly created based on software components and a common architecture.

The explicit use of software architectures in the system development process has been applied in different works [Kruchten, 91] [Bass *et al.*, 98]. Software architecture can be defined as a structure, or a set of structures, of the system that comprises software elements, the externally visible properties of those elements, and their relationships [Bass *et al.*, 98]. Considering the relevance of the software architecture, research in several directions has been conducted. One of these directions is the proposal, representation, and use of reference architectures. Reference architecture plays a dual role with regard to software architectures of tools for a given domain [Gallagher]: it generalizes and extracts common functions and configurations and provides a base for instantiating target systems that use that common base more reliably and cost effectively. Thus, a reference architecture refers to a set of architectural views containing knowledge, represented by, for instance, requirements, modules and interfaces, and providing a vocabulary of a given domain as it captures the essence of the architectures of a collection of systems of that domain.

Based on that, reference architectures for different domains have been proposed, such as those discussed in [Angelov *et al.*, 09] and [Oliveira *et al.*, 10a]. Reference architectures aggregate knowledge of a given domain to the system development; thus, the idea is to promote reuse of this knowledge in order to develop new systems for that domain. One example of a Reference Architecture is RefTEST [Nakagawa *et al.*, 07], which provides information about architectural styles and patterns, modules, and functional requirements for testing tools. The main goal of RefTEST is to instantiate this knowledge, together with specific requirements and the analysis/design models of the intended tool, in order to build a single architecture, *i.e.*, the architectural instance. These ideas have already been applied to the implementation of different testing tools [Ferrari *et al.*, 10] [Nakagawa *et al.*, 10].

2.1 Related Works

In the last years, some works discuss the relevance of reference architecture to the testing domain [Nakagawa *et al.*, 07] and present the contribution of reference architecture to the development of testing tools. Ferrari [Ferrari *et al.*, 10] presents how to apply the RefTEST to implement a tool for automating the mutation testing of aspect-oriented Java programs. Nakagawa [Nakagawa *et al.*, 10] proposed to apply a reference architecture to develop software for configuration management (SCM), and presents a case study to the development of a SCM tool for the software testing domain. Although these works present a relevant contribution to the testing domain, there is a lack of works that investigate how to apply reference architecture to derive testing tools from a SPL.

On the other hand, there are some works related to testing software product lines [Olimpiew, 05] [Engström, 11] [Oster *et al.*, 11]. For instance, the Engström work [Engström, 11] is focused on identifying the main challenges in SPL testing and also what topics for testing PL have been investigated in the last years. A survey on Model-Based Software Product Lines Testing that compares some MBT approaches to test SPL is found in [Oster *et al.*, 11] . Although there are several works that discuss software testing and SPL, all of them focus on the testing of software product lines and does not investigate how the SPL concepts can be applied to support the development of testing tools. Moreover, to the best of our knowledge, there is lack of investigation on the use of reference architecture and SPL to build testing tools.

3 PLeTs

PLeTs is a SPL to automate the generation of MBT tools. These tools aim to automate the test activities of MBT processes [Rodrigues *et al.*, 10] [Silveira *et al.*, 11]. The MBT process automates the generation of test cases and/or test scripts based on the system model. MBT tools derived from PLeTs support some or all of MBT activities (see Section 2). They accept a system model as an input, generate the test cases/scripts (*Expected Inputs/Outputs*), execute the test scripts and then compare the results. It is important to note that PLeTs was designed to generate scripts based on a template for a specific testing tool. As a consequence, a testing team could use its legacy testing tool to apply MBT, reducing effort and investment. After that, the PLeTs product runs the testing tool, loads the generated scripts, starts the test (*Run Tests*) and compares the results

(*Compare Results*). Thus, the PLeTs goal is the reuse of SPL artifacts (*e.g.*, models and software components) and use testing tools to make it easier and faster to develop a new MBT tool.

Furthermore, PLeTs also takes into account the knowledge contained in a Reference Architecture called RefTEST [Nakagawa *et al.*, 07] to generate MBT tools. RefTEST is a comprehensive Reference Architecture which involves many important concepts, such as, crosscutting concerns or architectural views. In this work, we apply only the RefTEST conceptual model for the core of the derived products, *i.e.* testing tools.

According to RefTEST, four main concepts are sufficient to represent the core elements of testing tools: *Test Artifact, Test Case, Test Criterion,* and *Test Requirement*. Moreover, each core element from RefTEST has associated the core test activities that a testing tool should contain [Nakagawa *et al.*, 10]. For instance, in the Table 1 the activity *Include test cases* is related to the concept *Test Case* while the activity *Generate test requirements* is related to *Test Requirement*.

Activities	**Concepts**	**Feature**
Acquire Test Artifacts	Test Artifact	Parser
Automatically Generates Test Cases	Test Case	Test Case Generation
Minimize Set of Test Cases	Test Case	Test Case Generation
Include Test Cases	Test Case	Test Case Generation
Insert drivers and stubs	Test Artifact	Script Generation
Remove Test Cases	Test Case	Test Case Generation
Remove Test Requirements	Test Requirement	Parser
View Test Requirements	Test Requirement	Test Case Generation
⋮	⋮	⋮
View drivers and stubs	Test Artifact	Script Generation

Tabela 1: Mapping Test Activities and Concepts to PLeTs Features

As RefTEST provides a well-established and consolidated set of the testing activities and concepts, such information was used when specifying the features of our SPL for MBT tools. Therefore, the MBT tools derived from PLeTs are composed of testing activities and requirements, as well as, input models, testing techniques, tools and test domains for a specific MBT tool. Table 1 presents the relationship between RefTEST concepts and PLeTs features.

Figure 1 presents the current PLeTs feature model, which is composed of four main parent features: *Parser, TestCaseGenerator, ScriptGenerator,* and *Executor*. It is important to mention that, even though our current feature model has a well-defined number of features, this model can, and will be expanded to include new features. A description of our SPL features is as follows:

- *Parser* automates the *Build Model* step in the MBT main activities. It is a mandatory feature with two child features, *Uml* and *Text*. The former extracts information from UML models and the latter extracts information from a textual file. As shown in Table 1, the *Parser* feature could also implement some testing activities from RefTEST, *e.g.*, the activity *Acquire Test Artifacts*;

- *TestCaseGenerator* represents the *Generated Expected Inputs* step in the MBT main activities. It is a mandatory feature with two child features: *FormalModel* and *AbstractTestCaseGenerator*. The former has two features: *FiniteStateMachine*, which has the child features *HSI* [Sabnani, 88] [Petrenko *et al.*, 93] and *UIO* [Anido, 95], and *PetriNets*. The latter has two child features *PerformanceTesting* and *StructuralTesting*. These two features receive the generated test sequence and create the abstract testing sequence to each testing level. As shown in Table 1, the *Test Case Generation* feature could also implement the *Include Test Cases* or *View Test Requirements* from the RefTEST main activities;

- *ScriptGenerator* is an optional feature that is used to convert abstract test cases into scripts for a testing tool responsible to perform the actual execution of the System Under Test (SUT). *ScriptGenerator* has two child features: *LoadRunnerScript* and *JMeterScript* that are used to consolidate the abstract test case in scripts to a specific testing tool;

- *Executor* represents the product interface and the test execution. This feature also has two child features: *ProductInterface* and *Parameterization*. The former defines the product interface and is composed of two child features: *GUI* to graphical interfaces and *Console* to command line execution. The latter has two child features: *JMeterParameters* and *LoadRunnerParameters*. These features are implemented to start the testing tool and to run the test scripts. After that, the testing tool (*e.g.*, LoadRunner) collects the test result, shows the results and compares them with test oracles.

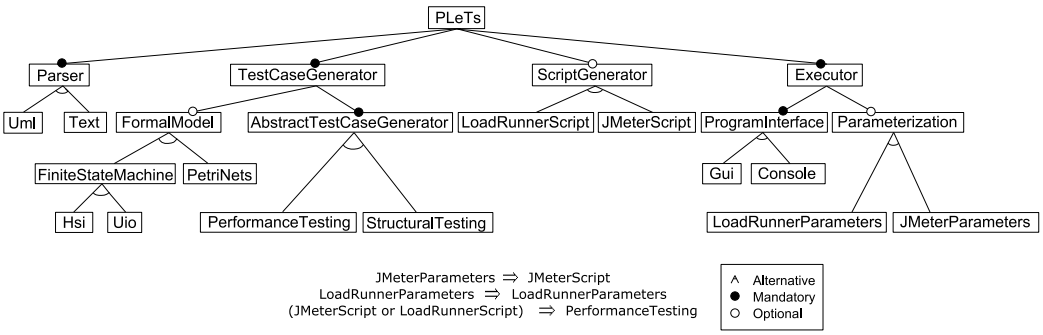


Figura 1: PLeTs Feature Model

As described in this section, several activities related to the concepts of RefTEST are implemented in the sub-features of PLeTs feature model. It is important to mention that activities from one specific concept can be spread through different implemented features, for example, *View Test Requirements* and *Remove Test Requirements*, from the *Test Requirements* concept in RefTEST are implemented, respectively, in the *Test Case Generation* and *Parser* features from PLeTs. As can be seen in Table 1, activities from different concepts in RefTEST can be implemented in the same feature of PLeTs.

With regard to Figure 1, there are several dependencies that are denoted by propositional logic between features. For instance, if feature *Executor* and its child feature *LoadRunnerParameters* are selected, then the feature *ScriptGenerator* and its child feature *LoadRunnerScript* must be selected as the generated tool is not able to execute tests with no test script. Furthermore, it is important to note that the PLeTs feature model can be extended to support new testing techniques or tools by adding new child features to its main features. For instance, if one adds new features for the SilkPerformer testing tool, new child features for the *ScriptGenerator*, *ProductInterface* and *Parameterization* features must be included.

Therefore, when test engineers want to add a new feature to PLeTs, they have to consider whether the new feature will have any dependency on the existing features, or whether they will have to implement new features. The latter is not desirable. This is important because different software testing tools may need different system models, formal models, scripts generation, and execution. For example, if engineers want to generate a new product to test parallel systems, they can reuse the parser and the test case generation already developed, but they might need a new feature able to generate the test scripts for a specific tool to test parallel systems.

4 PLeTs Architecture and Implementation

As mentioned in the Section 3, PLeTs is designed based on two main concepts: Software Product Line and Reference Architectures. From SPL we apply both reactive and extractive approaches. From Reference Architectures we apply RefTEST. In order to implement PLeTs, we have used a plugin-based mechanism to develop each feature of our feature model [Cervantes, 06].

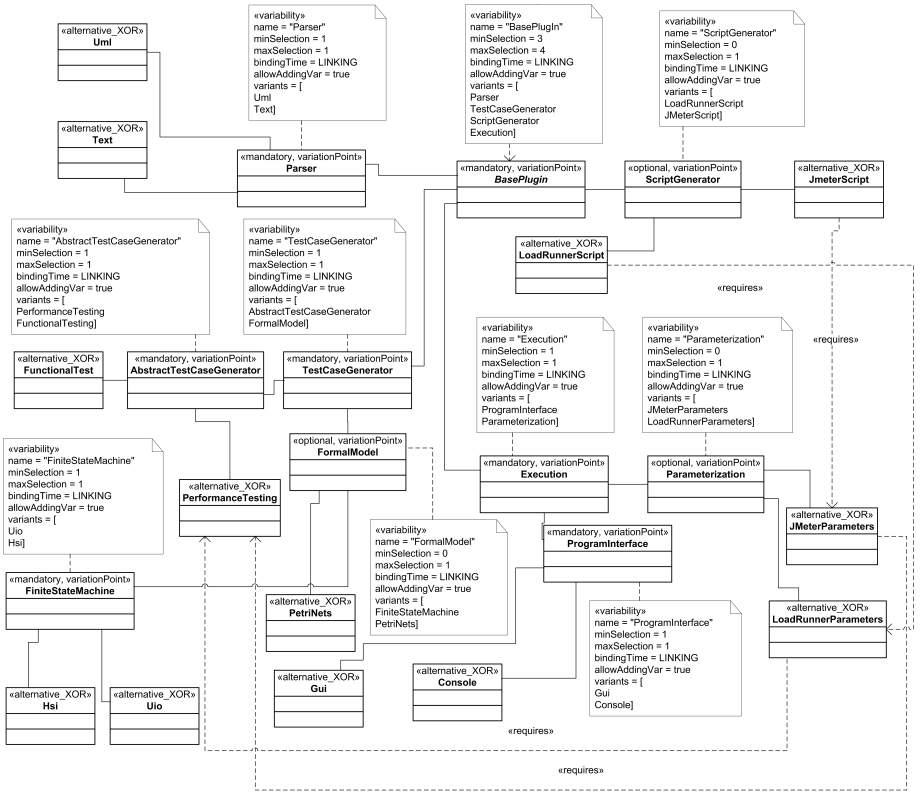


Figura 2: PLeTs UML class diagram with SMarty

The main idea of this strategy is to plug a component without having to have special explicit intertwined configuration of a specialized software engineer. The combination of plugins, one for each desirable feature, generates a product. Thus, a MBT tool derived from PLeTs is assembled by installing a set of selected plugins on a common software base. We chose this approach to generate the PLeTs products because it presents several advantages, as the high-level of modularity and decoupling between the base application and plugins. Furthermore, a plugin-

based SPL presents other benefits as, for instance, the plugins can be developed independently and geographically distributed, reducing time to market and costs [Cervantes, 06].

To manage the dependencies among plugins and represent the variability in PLeTs, we apply the SMarty (**S**tereotype-based **M**anagement of V**ar**iabili**ty**) approach [Oliveira *et al.*, 10b]. SMarty is composed by a UML profile and a process for managing variabilities in a PL. The SMarty profile contains a set of stereotypes and tagged values to denote the PL variability. The SMarty process consists of a set of activities that guide the user to trace, identify, and control variabilities in a PL. Figure 2 shows the PLeTs class model in accordance to SMarty that reflect our feature model shown in Figure 1. Although our PL has several components and each component has a set of classes, with readability purpose, in the Figure 2 we represent only the classes that implement the component interface. The main PLeTs components are: *BasePlugin, Parser, TestCaseGenerator, ScriptGeneration* and *Executor*.

- *BasePlugin* is a mandatory variation point component that has four variants, in which three are mandatory components, *Parser*, *TestCaseGenerator* and *Executor* and one component, *ScriptGenerator*, is optional. Its associated variability, denoted by tag ≪*variability*≫, indicates that its minimum number of variants is two (*minSelection = 3*) and the maximum is four (*maxSelection = 4*). In addition, the new variants could be included at linking time (*bindindTime = LINKING*).

- *Parser* is both a mandatory variant and a variation point that has two exclusive variants components, *Uml* and *Text*. Its associated variability indicates that its minimum number of variants is one (*minSelection = 1*) and the maximum is 1 (*maxSelection = 1*).

- *TestCaseGenerator* is both a mandatory variant and a variation point that has two variants components: the optional component *FormalModel* and the mandatory component *AbstractTestCaseGenerator*. The former has two exclusive components *PetriNets* and *FiniteStateMachine*, which has the exclusive variant components *Uio* [Anido, 95] and *Hsi* [Sabnani, 88]. The latter has two exclusive variant components, *PerformanceTesting* and *StructuralTesting*.

- *ScriptGeneratorPlugIn* is both an optional variant and a variation point that has three exclusive variant components, *LoadRunnerScript* and *JmeterS-*

cript. Thus, its minimum number of variants is one (*minSelection = 1*) and the maximum is 1 (*maxSelection = 1*).

- *Executor* is both a optional variant and a variation point that has two variants components, the mandatory *ProgramtInterface* and the optional *Parameterization*. The former has two exclusive variant components, *Gui* and *Console*. The latter has three exclusive variant components *LoadRunnerParameters* and *JmeterParameters*.

In all components, apart from *BasePlugin*, each associated variability indicates that exclusively one of the variants can be selected, as well as new variants could be included at linking time. The SMarty approach allows to represent situations in which the selection of a variant forces the selection of another variant, as a constraint among the variants. For instance, if one selects the component *LoadRunnerParameters* to compose a PLeTs product, it requires that the component *LoadRunnerScript* is selected. The latter component requires the class *PerformanceTesting*.

It is important to notice that the constraints presented in the variability class model will be used as input by the PlugSPL environment to resolve the dependencies among features [PLeTs PL]. Moreover, PlugSPL is used to design the PLeTs feature model, generate the PL architecture and support the generation of PLeTs products [Rodrigues *et al.*, 12]. One example of a product derived from PLeTs could have the following components: *BasePlugin*, *Uml*, *Parser*, *TestCaseGenerator*, *AbstractTestCaseGenerator*, *PerformanceTesting*, *FormalModel*, *FiniteStateMachine*, *Hsi*, *ScriptGenerator*, *JmetersScript*, *Execution*, *Parameterization*, *JMeterParameters*, *ProgramInterface* and *Console*. It is important, also, to mention that every node on the feature model is a feature, and every feature is a component - one-to-one mapping.

4.1 Example: A MBT Tool for Web Applications

This section describes how to derive a performance product variant from PLeTs. As mentioned at the beginning of Section 4, PLeTs is designed and developed using extractive and reactive approaches. Thus, based on the features already implemented, PLeTs incrementally grows as the demand for new software arises. In this section, we show how to build a new product to execute performance test for web applications. First of all, we worked together with the testing team of a big IT company (more than 20.000 employees) to define the requirements

they had for a set of web applications they wanted the new tool to test. The basic requirements were: a) the MBT tool should use, as input, UML models already developed during the software development process; b) the MBT tool should be able to generate test scripts for the Visual Studio (VS) performance testing tool [Guckenheimer, 06]; c) the MBT tool should be able to automatically execute the performance testing using VS.

Based on the above requirements, we analyzed the set of features that we had already developed to see whether we could reuse some of them and which new ones we should develop. Our analysis showed that we could reuse some features from our model (*Uml, Parser, TestCaseGenerator, AbstractTestCaseGenerator, PerformanceTesting, FormalModel, FiniteStateMachine, Hsi*, *ScriptGenerator, Executor, Parameterization, ProgramInterface* and *Console*) and that we had to develop some new child features for the *ScriptGenerator* and *Parameterization* features. These new child features are called *VisualStudioScript* and *VisualStudioParameters* respectively and they were included in our PLeTs feature model and were developed as plugins.

The new class diagram for a performance tool is shown in Figure 3. This class diagram is an instance of the PLeTs class model presented in Figure 2. As can be seen in Figure 3, only the components *BasePlugin, Uml, Parser, TestCaseGenerator, AbstractTestCaseGenerator, PerformanceTesting*, *FormalModel, FiniteStateMachine, Hsi, ScriptGenerator, VisualStudioScript, Execution, Parameterization, VisualStudioParameters, ProgramInterface* and *Console* are present for this new MBT tool - PLeTsPerf.

The MBT tools should perform the following basic activities based on information extracted from models: generate test cases, generate scripts and execute scripts using a testing tool. Furthermore, the MBT approach uses the system model as an input to generate test cases/scripts. In our example, the PLeTsPerf uses, as input, UML diagrams with stereotypes. These diagrams are represented in an XMI file used to generate test cases/scripts. Before the test cases/scripts are generated, the XMI file is parsed and converted into a formal model, *e.g.*, Finite State Machine (FSM). The FSM is used as input to execute the HSI Method [Petrenko *et al.*, 93]. The HSI Method generates the sequences of activities that have to be executed.

As mentioned above, some performance stereotypes could be added to the UML models. In our strategy, when using UML models, stereotypes are the base to include the necessary information to generate our test cases/scripts. To generate the tool described in this section we have included performance stereotypes in two UML diagrams: *Use Case* and *Activity*. The performance stereotypes are

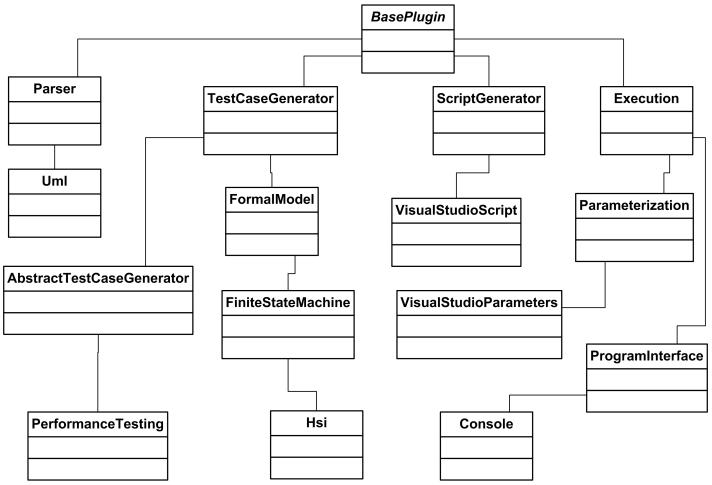


Figura 3: UML Class Diagram - Performance testing product that uses VS

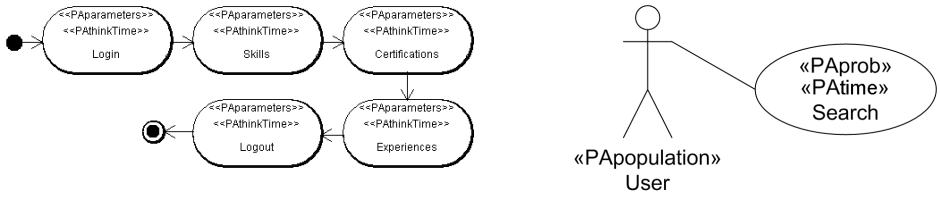
the following: a) ≪PApopulation≫: this stereotype has two tags: the first one represents the number of users that are running the application, while the second one represents the host where the application is executed (defined in all actors of the use cases diagram); b) ≪PAprob≫: defines the probability of execution for each existing activity; c) ≪PAtime≫: expected time to perform a given use case; d) ≪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; e) ≪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 previous works did not include).

Taking these stereotypes into consideration, the UML parser plugin extracts the information from the UML Models. Based on these information a FSM is generated and the HSI method is applied to generate the test sequences. Then, the plugins *PerformanceTesting* creates the performance abstract test cases. After the generation of the abstract test cases, the plugin *VisualStudioScript* will use the information contained in the abstract test cases to generate test scripts to VS. After that, the plug-in *VisualStudioParameters* will automatically start the VS, load the test scripts and perform the test in the SUT.

5 Case Study: Skills Management Tool

In this section, we apply the performance tool derived from PLeTs to an application that manages 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 a team of an IT company. The Skills tool was developed in Java, using the MySQL database for data persistence and Tomcat as web application server.

The application is modeled using UML diagrams augmented by the performance stereotypes presented in Section 4. One example of our use of UML with stereotypes is the *Search* case. Figure 4 (b) shows part of the user interaction behavior with Skills. Furthermore, the necessary steps to implement this use case is detailed in the activity diagram shown in Figure 4 (a). 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 certifications assigned to the actor; *Experiences* to list the user's professional experience; and *Logout* to exit the system.



(a) (b)

Figura 4: Skills UML Models annoted with performance stereotypes

Once all the UML diagrams (*e.g.*, see Figures 4 (a) and 4 (b)) have been constructed, we use PLeTs to derive a tool to generate the test scripts for the Skills Management Application. The scripts were initially generated to run on LoadRunner, but we have changed the plugin that generates scripts and then applied the same set of test scripts using a different testing tool (Visual Studio). As described in Section 4.1, we can include six stereotypes in the UML diagrams, with one or more tags. As can be seen in Figure 4 (b), the *Search* use case diagram has three of those stereotypes, and they are generated[2] with the following values:

```
PApopulation
```

[2]The values for the stereotypes can be generated automatically using different generation strategies, *e.g.*, based on log files or randomly.

```
   Url = "http://localhost/skillsApp/mainHome/"
   InitialUsers = "25"
PAprob
   Prob = "1.0"
PAtime
   RunDuration = "600"
```

Another example of stereotypes that are included in the activity diagram from Figure 4 (a), are shown in the *Login* activity and has the following values:

```
PAthinkTime
   ThinkTime = "3"
PAparameters
   storage = "c:\users\...\skills.webtest"
   Parameter Name = "Name" Value = "user.name"
   Parameter Name = "Pass" Value = "user.password"
   Percentage = "100"
   Method="POST"
```

Notice that the *Parameter Name* tag is the concatenation of two pieces of information: *Name* and *Pass*. These tags are extracted from the UML diagram and processed by the PLeTs tool plugin that generates scripts for VS. Once the test script generation is completed, the derived performance tool calls the VS tool and automatically starts the test. It is important to highlight that we have used the default VS standard template. We could have redefined other information in the template, but for the Skill Management application this was not necessary. Figure 6 shows an extract from the template that was generated for the LR tool and Figure 5 shows the template that was generated for the VS tool. These templates show the actions of an user, for example, the *think time* (*ThinkTime = "3"*) and the parameters *username* and *password*.

As presented in Section 4.1, the development of the performance testing tool that uses VS was supported by the development of two new plugins, *VisualStudioScript* and *VisualStudioParameters*, and the reuse of other plugins, *BasePlugin, Uml, Parser, TestCaseGenerator, AbstractTestCaseGenerator, PerformanceTesting, FormalModel, FiniteStateMachine, Hsi, ScriptGenerator, Execution, Parameterization, ProgramInterface* and *Console*, that were previously developed for another performance tool, *i.e.*, LoadRunner. This showed that to derive a new product it is not necessary to develop all plugins that are needed from scratch. Another important aspect, is that the set of plugins that can be selected to compose a product can be easily modified to support new features. Based on that, it is possible, for a company that is using the VS performance tool, to apply model-based testing in their testing process and change the monitored performance metrics through the modification of the plugins *VisualStudioScript* and *VisualStudioParameters*.

```
<WebTest Name="Login"...>
 <Items>
  <TransactionTimer>...</TransactionTimer>
  <TransactionTimer Name="Login">
   <Items>
    <Request Url="http://localhost/
    skillsApp/mainHome/"
    ThinkTime="3"...>
     <QueryStringParameters>
       <QueryStringParameter Value=
       "{{$user.name}}" Name="name".../>
       <QueryStringParameter Value=
       "{{$user.password"
       Name="pass".../>
     </QueryStringParameters>
    </Request>
   </Items>
  </TransactionTimer>
 </Items>
 <ValidationRules>...</ValidationRules>
</WebTest>
```

Figura 5: Test script generated - VS

```
Action()
{
...
lr_think_time(3);
web_submit_data("Login.jsp",
"Action=
http://localhost/skillsApp/
mainHome/",
"Method=POST",
"RecContentType=text/html",
"Referer=",
"Mode=HTML",
ITEMDATA,
 "Name=name", "Value=
 {{$user.name}}", ENDITEM,
 "Name=pass", "Value=
 {{$user.password}}", ENDITEM,
LAST);
...
}
```

Figura 6: Test script generated - LR

6 Lessons Learned

This section presents the lessons learned from the development of PLeTs, using a Reference Architecture, and also from the process to deriving a MBT tool from PLeTs.

As presented in Sections 1 and 2, the SPL adoption has increased in the past years since several companies report successful cases. As a consequence, SPL has become a well-known and wide-applied approach to promote reuse, minimize time-to-market and cost of software systems. Despite these advantages and the relevance of software testing, to the best of our knowledge, there is no academic or commercial work proposing a product line to derive model-based testing tools. Nevertheless, the development of a model-based testing PL, as PLeTs, provides many other benefits beyond those presented above. The most relevant benefits that we could identify while developing PLeTs are related to:

- the flexible way that features can be added to the PL to support a new functionality. In order to make the development of new features flexible, as well as, their integration in the PL we use the concept of plugins. Based on that, a new plugin can be developed from scratch and easily added to the PL. On the other hand, we can select a pre-existent plugin, modify it, if necessary, to support a different functionality, and then add it to the PL as a new

feature. However, when theses features (plugins) are added to the PL, it is necessary to manage their variability. Figure 2 (Section 4) represents this situation. In the figure, some plugins have a dependency relation with another plugin, denoted by $\ll$*requires*$\gg$. A not desired consequence of that is that the complexity to manage and represent the dependency grows along with PLeTs. To minimize this effect, when we designed the PL we decided that the dependency should be defined in each plugin. This approach simplifies the development of PLeTs, because the responsibility for managing the dependencies among plugins is an obligation of the developer of plugins. Despite the fact that our approach works well at the moment, we are not certain if the widespread use of dependency among plugins will become a limitation in the future;

- the use of a reference architecture to derive MBT testing products. The key issue to use a reference architecture is that it facilitates the reuse of design expertise by achieving solid, well-recognized understanding of a specific domain. Therefore, in our work we take into account a reference architecture for testing tools - RefTEST. The use of a reference architecture in PLeTs brought about many advantages, for instance, it eases the evolution and the maintenance of the generated testing tools. However, from our point of view, the most important advantage is that we can map, in an easy way, the testing activities into the main features of PLeTs, *i.e.*, *Parser*, *TestCaseGeenerator*, *ScriptGenerator* and *Executor*. Therefore, to develop a new plugin it is easy to know exactly what testing activities should be implemented;

- the fact that the MBT tools derived from PLeTs can manage the whole MBT process. The derived tools should be able to accept some form of SUT model as an input, and based on that generate an output from it. The output could be a test case suite or a script to a specific testing tool. However, we designed PLeTs bearing in mind that in most cases, a company that will adopt our MBT tool to conduct their testing process could already have a defined testing process and therefore some kind of testing tool. Furthermore, the design and development of a MBT tool from scratch is time consuming and a high cost activity. Because of that, we have designed/developed PLeTs to support the automatic creation and execution of the scripts/templates to academic or commercial testing tools. The only functionality required is that these testing tools import scripts or use some kind of template

files. It is important to highlight that as up to this moment we have already developed plugins to generate scripts and execute them on VisualStudio, LoadRunner and JMeter testing tools.

- evolution of the product line. Our product line has been used in the context of a collaboration with an IT company. Although our previous feature model [Rodrigues *et al.*, 10] [Silveira *et al.*, 11] had been used to generate new products and these products were applied to some actual applications, we noticed that most of the time the *Executor* feature, which was an optional feature, was selected because we needed some strategy to execute the *Parser* and *TestCaseGenarator* features. This situation, and some feedback from test engineers from the IT company, resulted in an evolution of our feature model to include the *Executor* feature as a compulsory feature. Basically, some test engineers would prefer to use a textual interface (*Console*) and others would rather use a graphical interface (*GUI*) (see Figure 1). Therefore, feature *Executor* was changed to mandatory and two sub-features were added: *ProgramInterface* and *Parameterization*. The former feature defines the user interface with the MBT product, and the latter feature defines the necessary parameters to use external testing tools, *e.g.*, LodRunner.

7 Conclusion

Software testing is a fundamental activity for assuring software quality. Nevertheless, it has a high cost when compared to the other stages of software development. Automation of software testing through reuse of software artifacts is a good alternative for mitigating these costs and making the process much more efficient and efficacious. MBT is a technique to automatic generation of testing artifacts based on software models. Because of that, a diversity of MBT tools has been developed in the last years. Despite that, the knowledge and the artifacts created for those tools cannot be fully reused.

Two approaches that can ease the above mentioned problems are Software Product Lines (SPL) and Reference Architectures. SPL has added the possibility of systematically generating a diversity of software products at lower costs, in shorter time, and with higher quality. Reference Architectures, on the other hand, have been playing a significant role in contributing to the success in the development of software systems.

The main contribution of this work was the use of a Reference Architecture

and SPL to generate testing tools that support MBT. We also presented the PLeTs tool and explained how to derive a new performance product from PLeTs (PLeTs-Perf). It is important to highlight that our SPL is designed to be comprehensive and not only to support the generation of performance testing tools, but also to support the generation of MBT tools for different domains and testing techniques, e.g., structural or functional testing. The initial achieved results point out the relevance of the Reference Architecture, as well as SPL, to improve productivity and reuse during the generation of testing tools. As future work, we are planing to perform an empirical experiment to know the effort to use our MBT performance testing tools when compared to a commercial performance tool.

Acknowledgments

We thank CNPq/Brazil, CAPES/Brazil, INCT-SEC, and Dell Computer for the support in the development of this work.

Referências

[Abbors *et al.*, 10] Abbors, F., Backlund, A., Truscan, D. "MATERA - An Integrated Framework for Model-Based Testing,"*IEEE International Conference on the Engineering of Computer-Based Systems*, vol. 1, pp. 321-328, 2010.

[Angelov *et al.*, 09] Angelov, S., Grefen, P., Greefhorst, D. "A classification of software reference architectures: Analyzing their success and effectiveness,"in *Proceedings of the 8th Working IEEE/IFIP Conference on Software Architecture*. Cambridge, UK: IEEE, pp. 141-150, 2009.

[Anido, 95] Anido, R., Cavalli, A. "Guaranteeing Full Fault Coverage for UIO-Based Testing Methods,"in *Proceedings of the 8th International Workshop for Protocol Test Systems*. Chapman & Hall, pp. 221-236, 1995.

[Avizienis *et al.*, 04] Avizienis, A., Laprie, J. C., Randell, B., Landwehr, C. "Basic concepts and taxonomy of dependable and secure computing,"*IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11-33, 2004.

[Bass *et al.*, 98] Bass, L., Clements, P., Kazman, R. *Software architecture in practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[Bertolino, 07] Bertolino, A. “Software testing research: Achievements, challenges, dreams,”in *Proceedings of the 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, pp. 85-103, 2007.

[Boudier *et al.*, 88] Boudier, G., Gallo, F., Minot, R., Thomas, I. “An overview of PCTE and PCTE+,”*SIGSOFT Software Engineering Notes*, vol. 13, pp. 248-257, 1988.

[Cervantes, 06] Cervantes, H., Charleston-Villalobos, S. “Using a Lightweight Workflow Engine in a Plugin-Based Product Line Architecture,”in *Proceedings of the 9th international conference on Component-Based Software Engineering*, Springer-Verlag, pp. 198-205, 2006.

[Clements, 01] Clements, P., Northrop, L. *Software product lines: practices and patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[Dalal *et al.*, 99] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., Horowitz, B. M. “Model-based testing in practice,”in *Proceedings of the 21st International Conference on Software Engineering*. New York, NY, USA: ACM, pp. 285-294, 1999.

[DeMillo *et al.*, 78] DeMillo, R. A., Lipton, R. J., Sayward, F. G. “Hints on test data selection: Help for the practicing programmer,”*Computer*, vol. 11, pp. 34-41, 1978.

[El-Far, 01] El-Far, I. K., Whittaker, J. A. *Model-based Software Testing*. New York, NY, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[Engström, 11] Engström, E., Runeson, P. “Software product line testing - a systematic mapping study,”*Information and Software Technology*, vol. 53, pp. 2-13, 2011.

[Feng *et al.*, 07] Feng, Y., Liu, X., Kerridge, J. “A product line based aspect-oriented generative unit testing approach to building quality components,”in *Proceedings of the 31st Annual International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, pp. 403-408, 2007.

[Ferrari *et al.*, 10] Ferrari, F. C., Nakagawa, E. Y., Rashid, A., Maldonado, J. C. "Automating the mutation testing of aspect-oriented Java programs,"in *Proceedings of the 5th Workshop on Automation of Software Test*. New York, NY, USA: ACM, pp. 51-58, 2010.

[Gallagher] Gallagher, B. P. "Using the architecture tradeoff analysis method to evaluate a reference architecture: A case study,"available at: http://www.sei.cmu.edu/reports/00tn007.pdf, CMU/SEI, Tech. Rep.

[Gao *et al.*, 97] Gao, J., Chen, C., Toyoshima, Y., Leung, D. K. "Developing an integrated testing environment using the world wide web technology,"in *Proceedings of the 21st International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, pp. 594-601, 1997.

[Garlan, 00] Garlan, D. "Software architecture: a roadmap,"in *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, pp. 91-101, 2000.

[Grosskurth, 05] Grosskurth, A., Godfrey, M. W. "A reference architecture for web browsers,"in *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, pp. 661-664, 2005.

[Guckenheimer, 06] Guckenheimer, S., Perez, J. J. *Software Engineering with Microsoft Visual Studio Team System*. Boston, MA, USA: Addison-Wesley Professional, 2006.

[Harrold, 00] Harrold, M. J. "Testing: a roadmap,"in *Proceedings of the Conference on the Future of Software Engineering*. New York, NY, USA: ACM, pp. 61-72, 2000.

[Huima, 07] Huima, A. 'Implementing conformiq Qtronic,"in *Proceedings of the 19th International conference, and 7th international conference on Testing of Software and Communicating Systems*.Springer-Verlag, Berlin, Heidelberg, 1-12, 2007.

[Jing *et al.*, 10] Jing, Y., Lan, Z., Hongyuan, W., Yuqiang, S., Guizhen, C. "JMeter- based aging simulation of computing system,"in *Proceedings of the In- ternational Conference on Computer, Mechatronics, Control and Electronic Engineering*. Changchun, China: IEEE, pp. 282-285, 2010.

[Kruchten, 91] Kruchten, P. “Un Processus de Développement de Logiciel Itératif et Centré sur l´Architecture (An Iterative Software Development Process Centered on Architecture),”in *4éme Congrás de Génie Logiciel*. Washington, DC, USA: IEEE Computer Society, pp. 369-378, 1991.

[Krueger, 01] Krueger, C. W. “Easing the transition to software mass customization,”in *Proceedings of 4th International Workshop on Software Product-Family Engineering*. London, UK: Springer-Verlag, pp. 282-293, 2002.

[LoadRunner] Hewlett Packard - HP, “Software HP LoadRunner,”Available in: https://h10078.www1.hp.com/cda/hpms/display/main/hpms/content.jsp.

[Ma *et al.*, 06] Ma, Y. S., Offutt, J., Kwon,Y. R. “Mujava: a mutation system for java,”in *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: ACM, pp. 827-830, 2006.

[Matsumoto, 07] Matsumoto, Y. “A guide for management and financial controls of product lines,”in *Proceedings of the 11th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, pp. 163-170, 2007.

[Myers, 79] Myers, G. J. *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.

[Mayer *et al.*, 03] Mayer, J., Melzer, I., Schweiggert, F. “Lightweight plug-in-based application development,”in *International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. London, UK: Springer-Verlag, pp. 87-102, 2003.

[Misura *et al.*, 05] Misurda, J., Clause, J. A., Reed,J. L., Childers, B. R., Soffa, M. L. “Demand-driven structural testing with dynamic instrumentation,”in *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM, pp. 156-165, 2005.

[Nakagawa *et al.*, 11] Nakagawa, E. Y., Ferrari, F., Sasaki, M. M. F. “An Aspect-Oriented Reference Architecture for Software Engineering Environments,”*Journal of Systems and Software*, vol. 84, pp. 1670-1684, 2011.

[Nakagawa *et al.*, 07] Nakagawa, E. Y., Simao, A. d. S., Fabiano, F., Maldonado, J. C. "Towards a reference architecture for software testing tools,"in *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering*. Boston, MA, USA: Knowledge Systems Institute Graduate School, pp. 1-6, 2007.

[Nakagawa *et al.*, 10] Nakagawa, E. Y., Trevisan, J. V. T., Maldonado, J. C. "Software configuration management as a crosscuttting concern: An example on testing,"in *Proceedings of the 22th International Conference on Software Engineering and Knowledge Engineering*. Redwood City, CA, USA: Knowledge Systems Institute Graduate School, pp. 483-488, 2010.

[Northrop, 02] Northrop, L. M. "SEI's software product line tenets,"*IEEE Software*, vol. 19, pp. 32-40, 2002.

[Olimpiew, 05] Olimpiew, E. M., Gomaa, H. "Model-based testing for applications derived from software product lines,"in *Proceedings of the 1st international workshop on Advances in model-based testing*. New York, NY, USA: ACM, pp. 1-7, 2005.

[Oliveira *et al.*, 10a] Oliveira, L. B. R., Felizardo, K. R., Feitosa, D. Nakagawa, E. Y. "Reference Models and Reference Architectures Based on Service-Oriented Architecture: A Systematic Review,"in *Proceedings of the 4th European Conference on Software Architecture*. Berlin, Heidelberg: Springer-Verlag, pp. 1-10, 2010.

[Oliveira *et al.*, 10b] Oliveira, E. A., Gimenes, I. M. S., Maldonado, J. C. "Systematic management of variability in UML-based software product lines,"*Journal of Universal Computer Science*, vol. 16, pp. 2374-2393, 2010.

[Oster *et al.*, 11] Oster, S., Wubbeke, A., Engels, G., Schurr, A. "A Survey of Model-based Software Product Lines Testing,"in *Model-based Testing for Embedded System*. CRC Press/Taylor, pp. 339-381, 2011.

[Peralta *et al.*, 08] Peralta, K., Orozco, A. M., Zorzo, A. F., Oliveira, F. M. "Specifying Security Aspects in UML Models,"in *Proceedings of 1st International Workshop on Modeling Security In ACM/IEEE 11th International Conference on Model-Driven Engineering Languages and Systems*. Toulouse, France: ACM, pp. 1-10, 2008.

[Petrenko *et al.*, 93] Petrenko, A., Yevtushenko, N., Lebedev, A., Das, A. "Nondeterministic State Machines in Protocol Conformance Testing,"in *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems*. Amsterdam, The Netherlands, Netherlands: North-Holland Publishing Co., pp. 363-378, 1993.

[PLeTs PL] PLeTs Product Line, " PLeTs Product Line,"available at: http://www.cepes.pucrs.br/plets/.

[Rapps, 85] Rapps, S., Weyuker, E. J. "Selecting software test data using data flow information,"*Transactions on Software Engineering*, vol. 11, pp. 367-375, 1985.

[Rodrigues *et al.*, 10] Rodrigues, E. M., Viccari, L. D., Zorzo, A. F. "PLeTs-Test Automation using Software Product Lines and Model Based Testing,"in *Proceedings of the 22th International Conference on Software Engineering and Knowledge Engineering*. Redwood City, CA, USA: Knowledge Systems Institute Graduate School, pp. 483-488, 2010.

[Rodrigues *et al.*, 12] Rodrigues, E. M., Zorzo, A. F., Oliveira, E. A., Gimenes, I. M., Maldonado, J. C., Domingues, A. R. "PlugSPL: An Automated Environment for Supporting Plugin-based Software Product Lines,"in *Proceedings of the 23th International Conference on Software Engineering and Knowledge Engineering*. Redwood City, CA, USA: Knowledge Systems Institute Graduate School, pp. 647-650, 2012.

[Sabnani, 88] Sabnani, K., Dahbura, A. "A protocol test generation procedure,"*Computer Networks and ISDN Systems*, vol. 15, pp. 285-297, 1988.

[Saudrais, 11] Saudrais, S., Chaaban, K. "Automatic relocation of AUTOSAR components among several ECUs,"in *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. New York, NY, USA: ACM, pp. 199-244, 2011.

[Silveira *et al.*, 11] Silveira, M. B., Rodrigues, E. M., Zorzo, A. F., Vieira, H., Oliveira, F. "Generation of Scripts for Performance Testing Based on UML Models,"in *Proceedings of the 23rd International Conference on Software Engineer- ing and Knowledge Engineering*. Miami, FL, USA: Knowledge Systems Institute Graduate School, pp. 1-6, 2011.

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

[Stantchev, 09] Stantchev, V. "Performance evaluation of cloud computing offerings,"in *Proceedings of the Third International Conference on Advanced Engineering Computing and Applications in Sciences*. Washington, DC, USA: IEEE Computer Society, pp. 187-192, 2009.

[Stefanescu, 09] Stefanescu, A., Wieczorek, S., Kirshin, A. "MBT4Chor: A Model- Based Testing Approach for Service Choreographies,"in *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*. Berlin, Heidelberg: Springer-Verlag, pp. 313-324, 2009.

[Utting, 06] Utting, M., Legeard, B. *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[Veanes *et al.*, 08] Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L. "Model-based testing of object-oriented reactive systems with Spec Explorer,"in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, vol. 4949, pp. 39-76, 2008.

[Whittaker, 00] Whittaker, J. A. "What Is Software Testing? And Why Is It So Hard?", *IEEE Software*, vol. 17, pp. 70-79, 2000.

[Yoshimura *et al.*, 06] Yoshimura, K., Ganesan, D., Muthig, D. "Assessing merge potential of existing engine control systems into a product line,"in *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*. New York, NY, USA: ACM, pp. 61-67, 2006.

[Young, 05] Young, M., Pezze, M. *Software Testing and Analysis: Process, Principles and Techniques*. New York, NY, USA: John Wiley & Sons, Inc., 2005.