Software Product Lines
System Test Case Tool: A Proposal

Crescencio Rodrigues Lima Neto, Ivan do Carmo Machado, Paulo Anselmo Mota Silveira Neto, Silvio Romero de Lemos Meira

Abstract—Nowadays the decision whether to work with Software Product Lines (SPL) or not becomes a binding requirement for the strategic planning of companies. Learning how to choose the ideal tools to test a SPL is beneficial for companies in this planning process. Even though the ascending number of available software engineering testing tools, SPL testing lacks specific tools capable of supporting the SPL Testing Process and managing the variability of test assets. This paper evaluates how to reduce the effort during the SPL Testing Process and consequently, how to make the variability of test assets manageable. We propose a software product line testing tool to build system tests from use cases that addresses challenges for SPL Testing we identified in a literature review.

Keywords—Software Testing; Software Product Lines; Software Reuse; Testing Tools;

I. INTRODUCTION

Software testing tools are available for testing in every stage of the software development life cycle, although it is not commercially viable for the tools vendors to produce a tool that suits every organization. It is inevitable, then customizations of tools is desirable [1]. Software product lines testing tools are not an exception, choose testing tools suitable for test applications and support process could be one of the most critical tasks of a project. In the SPL context, the amount of available tools decrease drastically, and the need of tools to reduce the effort during the SPL testing process is an obvious gap that need to be filled.

Software product lines is a planned, systematic and proactive reuse strategy, through exploiting the similarities within a set of products. SPL can enable organizations to achieve significant reductions in terms of development and maintenance cost and time-to-market as well [2], and remarkable quantitative improvements in productivity, quality and customer satisfaction [3], thus addressing the problems aforementioned.

In order to manage the variability, avoid the explosion of test cases due to the great number of variation points combinations and reduce the effort to test a software product line we need to implement testing tools that would allow for improvements in costs, time-to-market and quality [4].

The availability of tools makes testing a more systematic activity and can minimize the cost and time consumed, as well as the errors caused by human intervention [5]. A wide range of tools, with both commercial and academic purposes can be found. However, these tools have almost always been implemented for specific purpose and they are isolated from others, presenting its own architecture and internal structures [5]. As a consequence, difficulties in integration, evolution, maintenance, and reuse of these tools are very common. These tools often focus on automating specific testing techniques and criteria, without considering the whole testing process [5].

In this paper, we propose a SPL Test Tool for create system test cases based on use cases that supports the RiPLE-TE [6] test discipline of the RiPLE project. The RiPLE project is an effort of developing a framework to support a set of disciplines that compose the life-cycle of a SPL, namely Scoping [7], Requirements [8], Design [9], Implementation [10], Testing [11], Evolution Management [12].

We also investigate tools that can help during all tests levels of the SPL phases. Furthermore, in order to gure out the needs of the research field, we split our investigation into two steps. Firstly we analyzed test tools that have been developed for testing single systems; and secondly, we focused on tools that have been developed specifically to product lines. The question that motivates our work is: How to handle variation points and their combination within a test case?

The remainder of this paper is organized as follows. Section II describes the related work. Section III is an introduction to software product lines testing tools. Section IV analyzes the challenges of SPL Testing. Section V discusses the proposal. Finally, Section VI summarizes and concludes this paper.
II. RELATED WORK

There are few studies describing and detailing tools for testing SPL projects. If we decide to narrow the scope, encompassing the search for only system test related tools, within the SPL context, the findings are worse.

According to [13] the ScenTED-DTCD (Domain Test Case Scenario Derivation) is a prototype tool focused on generating test cases scenarios that describe the test engineer’s activities and the systems response if modeled within the activity diagram.

Another prototype tool can be found [14]. It generates system test scenarios from use cases. The approach is based on the automation of the generation of application system tests, for any chosen product, from the system requirements of a product line.

These studies are deemed to be good sources of information regarding tools for testing SPL. In order to develop our work, we considered every mentioned study, since they bring relevant information that comprises the knowledge to develop SPL system testing tools.

III. SPL TESTING TOOLS

Testing is an expensive and laborious phase of the software process [15]. In SPL, this assumption continues to hold true. As a result, testing tools were among the rst software tools to be developed. These tools now offer a bunch of facilities and their use can signiﬁcantly reduce the costs of testing [15].

Finding an effective and efficient software testing tool could be a life-saver for a project or a company. There is no single test tool suitable for all possible systems and industry sectors. Deciding what criteria to apply when selecting a speciﬁc tool for a project is quite tricky [16].

A software product line is a large system, with large volume of source code, and such automated tools can aid developers and testers to scan through the large volume of product lines source code and reveal unreliable program features to the human program. Testing tools can support testing large size of software product lines to achieve its goals [17].

An important factor in SPL Testing is automated testing and the use of testing tools. This decreases the effort when reusing test assets and makes the complex testing process more manageable [18]. There are numerous mature testing tools available. Despite the seeming abundance of available testing tools SPL testing lacks efﬁcient tool support. The problem with testing tools is that few of them directly support SPL [18].

A. Literature Review

This section presents how we identiﬁed papers reporting single systems testing tools and product lines testing tools, how we extracted data and information from the research papers, and how we analyzed them with respect to identify the requirements to compose a new testing tool focused on SPL Testing.

In order to extend the research and better understand the needs of tools for SPL Testing, we performed a research that Firstly identiﬁed 26 relevant studies using deﬁned search items related with the subject SPL testing tools. Then, we rened the search by applying some exclusion criteria to the study title. Next, we excluded studies on the basis of exclusion criteria applied to abstract and conclusion reducing the number to 15 papers. Finally, we obtained the primary studies set to be critically appraised. This set comprised 8 papers, that accordingly describe the implementation of a SPL testing tool.

In accordance with our ﬁndings, it is possible to check a huge amount of tools that support testing, but very few support SPL testing.

IV. SOFTWARE PRODUCT LINES TESTING CHALLENGES

SPL Testing may be represented as two instances of the V-model, according to [6]. First, the domain assets are tested. Then, the products are tested according to the model. This raises two problems. First, complete integration and system testing in domain engineering is usually not yet speciﬁed [18]. Try to test the different combinations of components leads to exponential growth of tested conﬁgurations [19]. Second, it is hard to decide on how much we can depend on the domain testing. According to [18] it does not seem to be clear where testing belongs in the overall SPL process.

Another challenger is the organization of test assets. Management of testing becomes very difﬁcult if testers cannot tell the difference between assets that belong to the domain and assets that belong to a product. Commonality and variability in product lines require rich traceability, from requirements to implementation and test assets, encompassing the whole development life cycle [19].

Integrating test environments and the test asset repository is a cumbersome work, that is even worse due to the lack of automated support tool, which leads to be a manually performed task. There is a need for speciﬁc testing tools that should help to manage the reusable testing assets and automate the execution of tests and the analysis of their results as well [18].

It is also evident the need of tools that integrate the whole process of software product lines testing including test environments and a test asset repository. A feasible approach for this would be to use a product lines speciﬁc integrated testing environment [20].

The product line approach requires a carefully planned testing process that can be easily adapted and used in different domains. Currently, there is a gap between the overall product line engineering processes and the assumed testing processes using the practical testing methods, because it is not clear how the processes and test assets are aligned to test
In order to lessen some of these problems, this work is focused on the building of a tool to support testing activities, at the system test level, aiming at reducing the required effort. It builds on the definition of a structured process for testing SPL, the RiPLE-TE [6], as next described.

V. TOWARDS A TOOL TO SUPPORT SPL TESTING

In RiPLE-TE, test assets are built in parallel to the development assets. Thus it is possible to maintain traceability among these artifacts, and consequently ease the process of developing and rebuilding them, if necessary. Testing a SPL includes the core asset, product specific assets, and their interactions.

In this effect, RiPLE-TE consists of two processes, considering the particularities of each phase, as showed in Figure 1. In Core Asset Development, when assets have to be developed with a special attention to the forthcoming reuse, the process advocates that unit and integration testing levels should be performed, whereas system and acceptance testing have to be postponed to Product Development, where the assets previously developed will be reused. Hence, knowledge produced in the core asset testing can be reused in product testing, reducing the overall effort.

Every test case created in the basis of the RiPLE-TE process can represent variability in a way that reuse is encouraged. Figure 2 shows an extract of a metamodel [21] developed in order to capture the relationship among the artifacts created in a SPL project. This figure basically depicts the Test model fragment, in which test cases are considered the main artifact to be handled.

Since each use case in a SPL project is supported by one or more test cases, the variability herein in handled as the same way than in use cases elaboration. This latter expands on the abstract use case definition, in which variability is represented in the use cases. A use case is herein composed by the entity AbstractFlow, that comprises the subentity Flow, which actually represent the use case steps. Every step can be associated with a subflow, that can represent a variation point. In both, fragments compose every (use or test) case, and are thus considered when deriving some product. This way, dependencies among test cases fragments and use case fragments make sense.

In addition the model also include other involved artifacts, such as planning and management assets.

A. Software Product Lines System Test Case Tool

The literature still lacks works describing tool support for testing software product lines [11], [17], [18]. In this effect, we propose a tool focused on the elaboration of system test cases for a SPL projects, thus encouraging reusable artifacts to be produced. We will use the test model previously mentioned (and fully described in [21]) to manage the test assets and dependencies.

Additionally, Figure 3 illustrates how the system test cases are built from use cases. According to [22], a use case goal can be decomposed recursively into sub-goals, as a result is necessary to create test cases for each fragments ensuring the use cases coverage.

1a. The use case document are composed by the use cases of the SPL project.
1b. The tool allow users to select all the use cases or part of these. The selected use cases will be extracted from the document described in 1a.
2a. Each use case can generate a system test case. When the use case is more complex the tool will generate a test cases group described below.
2b. Test Cases groups will be composed by two or more system test cases.
2c. The combination of system test cases generated by the tool will compose the Test Case Document. The tool also allow users to select specic test cases in order to generate customized test cases documents.
Next, the system test cases are formed by the following fields: an ID (unique identification number created automatically by the tool), the name of the test case, its summary and classification (positive and negative test cases), steps of the test case, pre and post conditions, variability type, use case references and screen path (describes the path of the system that should be tested).

When all the use cases were selected the application will focus at the construction of mandatory test cases. Thus the optional test cases can be built in accordance with the needs of specific products of the SPL product development phase. However, every mandatory use case, have to be validated, which consequently demands the creation of mandatory test cases. Besides test cases priority will be also classified in agreement with requirements priority defined by [8] as High, Medium and Low.

In addition to the proposal, test cases are composed by extracting information from use cases. Figure 4 illustrates the data extraction from the use case document (1x) for the construction of test case document (2x). The main flow (1y) of the use cases leads to the building of positive test cases (2y - when determined action should succeed [23]) that analyze if the main functionalities are working properly. Secondary flows (1z) of the use cases are divided in Alternative and Exception flows. Alternative secondary flows result in positive test cases (2z) that validate instructions that should succeed [23]. Finally, exception secondary flows result in negative test cases (2w - when determined action should not succeed [23]) that verify errors messages and unsuccessful procedures.

In order to manage the variability, all the variation points are associated with requirements specified in the SPL requirement document detailed by [21]. As requirements include variability, test cases must also contain explicit variability information. Test cases that include variability are called variant test cases. Test cases without variability are called common test cases and can be applied to all applications [13].

Moreover, each variation point are related with a use case, consequently for every single use case is possible to create test cases. Hence, variation points leads to the creation of test cases in this way preserving the variability within the tests cases.

The prioritization of the variation points is established in line with the variability type and priority of the requirements. Mandatory requirements with high priority have to be examined first. For this reason we propose that tests cases of mandatory variation points with high priority should be created first.

Figure 5 explains how the test case document can support the test architect. Using the document the test architect can delegate which tests should be created by developers (Unit and Integration tests because they need source code knowledge to support White-box techniques. White-box test cases consider the entire source code of the program while grey-box test cases only consider a portion of it [24]) and which tests must be done by test analysts (System and Acceptance tests because they need part or the entire system working, will be useful to support Blackbox techniques. Blackbox test cases are those in which no details about the implementation, in terms of code, is provided. These test cases are based exclusively on the inputs and outputs of the system under test [24]).

1) Tool Architecture: [21] presents a web tool implemented using the Django\textsuperscript{1} framework, which enabled the fast development of a functional prototype. Through Django, the metamodel mapped entities and their relationship into

\textsuperscript{1}http://www.djangoproject.com
Python\textsuperscript{2} classes, and then it is automatically created a relational database for these entities. Finally, Django generates a Web application where it is possible to test the mapping by inserting some test data, the documentation regarding features, requirements and so on.

Currently, it is possible to use the tool to document all the assets regarding the metamodel, however the test cases derivation from use cases is not supported yet [21]. This is the focus of our proposal and to bridge this gap we propose the architecture showed in Figure 6.

In order to understand the architecture of the proposed tool is necessary to know that Django follows the Web Server Gateway Interface (WSGI) protocol, so we can model incoming communication as browser and server. Figure 6 describes the architecture of the proposed tool combined with the SPL Metamodel. Details of the layers are described below:

- **Browser**: Users access the application through web browsers.
- **Template**: A Django template separates the application from the data. A template defines the presentation logic that regulate how the information should be displayed.
- **URL Dispatcher**: It defines which view is called for a given URL pattern. Basically, it is a mapping between URL patterns and the view functions that should be called for those URL patterns.
- **Model**: It describes the data structure/database schema. It contains a description of the database table, represented by a Python class. This class is called a model. Using it, is possible to create, retrieve, update and delete records in the database using simple Python code.
- **View**: It contains the business logic. View layer is itself a stack. Python view methods call a restricted template language, which can itself be extended with custom tags. Exceptions can be raised anywhere within a Python program, and if not handled they rise up as far as the environment.
- **Database**: All the data information will be recorded at the repository. Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and we have to make design decisions on which features to support and which assumptions we can make safely.

The view and the model layers can raise exceptions, but Python uses exceptions for control flow. So not all of these rise to the server.

The architecture of the proposed tool is suitable to support the construction of the test case document. However, to work managing source code is necessary to adapt the architecture, more details can be seen at Section VI.

**VI. CONCLUDING REMARKS AND FUTURE WORK**

Currently, despite of the increasing interest by the research community regarding SPL testing [11], it is very difficult to find suitable tools to support SPL testing processes. It has a direct impact in the high costs this activity poses, since testing becomes a cumbersome and laborious work [17].

This work proposes a system test tool to support the test process of a Software Product Line. The tool is aimed at reducing the effort of testing, by reducing the work required to follow a SPL testing process.

The creation of system tests cases will help to manage the traceability between use cases and tests cases. It is also necessary to organize and manage the variability of the tests assets after the creation of the test case document.

During the definition of our proposal we identify the necessity of extend the architecture in order to allow support the management of source code and subsequently automatically generate unit and integration test cases.

As a future work, we are intended to conduct a series of experimental evaluations in order to analyze the effectiveness of the proposed tool.

---

\textsuperscript{2}http://www.python.org
REFERENCES


