A general approach to Software Product Line testing

Elvira G. Ruiz[1], Jon Ayerdi[2], José A. Galindo[1], Aitor Arrieta[2], Goiuria Sagardui[2], and David Benavides[1]

[1] Universidad de Sevilla, Dept. Lenguajes y Sistemas Informáticos, Av. Reina Mercedes s/n Sevilla - España,
{egruiz, jagalindo, benavides}@us.es

[2] Mondragon Unibertsitatea, Dept. de Electrónica e Informática, Goiru 2, Mondragon - España
jon.ayerdi@alumni.mondragon.edu, aarrieta@mondragon.edu, gsagardui@mondragon.edu

Abstract. Variability is a central concept in Software Product Lines (SPLs). It has been extensively studied how the SPL paradigm can improve both the efficiency of a company and the quality of products. Nevertheless, this brings several challenges when testing an SPL, which are mainly caused by the potentially huge amount of products that can be derived from an SPL. Different studies proposing methods for testing SPLs exist. Furthermore, there are secondary studies reviewing and mapping the literature of the existing proposals. However, there is a lack of systematic guidelines for practitioners and researchers with the different steps required to perform a testing strategy of an SPL. In this paper, we present a first preliminary version for a tutorial that summarizes the existing proposals of the SPL testing area. To the best of our knowledge, there is no similar attempt in existing literature. Our goal is to discuss this tutorial with the community and enrich it to provide a more solid version of it in the future.

Keywords: Software product lines, Software testing, Software reusability.

1 Introduction

Software product lines and variability intensive systems benefits from a set of techniques, tools and methods that are used to develop a set of different products that share some commonalities [26]. The concrete functionality that varies across products in the SPL is encapsulated using an abstraction known as feature. Feature models are used to encode common and varying parts of SPLs [17]. In the literature, we find real examples encoding a large number of products. For example, the Linux Kernel [24] with more than 6,000 features or Debian packaging systems [11] with more than 27,000.

The large amount of products that an SPL can encode, makes its analysis a time–consuming and error prone task. Then, researchers proposed the use of

automated analysis techniques [5] for a set of activities in which testing is usually one of the most relevant [12].

SPL testing represents a new challenge for software testing practitioners and researchers [23]. When testing SPL, each product shares some common functionality with one or more products, while differing in at least one feature. SPLs add testing complexity because they require testing a set of products rather than a single product. These products, however, share common functionality or artifacts, enabling the reuse of some tests across the entire SPL.

According to [23], several strategies can be used to test SPL products. These testing strategies can be summarized as follows: *i*) testing product by product, *ii*) incremental testing, and *iii*) reusable asset instantiation. Testing product by product is a strategy that tests all products one by one, as if they were not part of an SPL. With this strategy the test process covers all possible interactions between features but grows exponentially in cost as a function of the number of features in the SPL. Incremental testing is a strategy that starts by testing the first developed product and creates new unit tests for each new feature added. Using this strategy, the commonalities in the SPL are exploited to reduce testing effort. However, when a new feature is introduced, all the interactions between the new feature and the old ones have to be tested, which can be challenging for large SPLs. Reusable asset instantiation relies on data captured in the domain analysis stage of SPL creation to develop a set of abstract test cases that cover all features (but not necessarily configurations) in the SPL. These abstract tests cases are mapped to concrete requirements in the application engineering stage. These last two testing strategies are designed to reduce the SPL combinatorial explosion in testing cost as a function of the feature count.

Within SPL engineering, two different processes can be distinguished: (1) domain engineering and (2) application engineering. Domain engineering is the process of developing the platform for building products and defining the commonalities and the variability of the product line. Application engineering is the process of deriving specific applications by using the platform defined in domain engineering and binding the variability to satisfy the needs of each particular application [22].

In [15] authors propose an ideal path to follow when it comes to SPL testing, which is a W testing model for SPLs that considers component, integration and system testing for both domain and application engineering. This paradigm maps every sub-process to either domain or application engineering. However, and due to the complexity of the testing processes when variability is considered, in many software projects there is not such a clear division of tasks. It is possible to find some testing processes (i.e., component testing) that can be started in the domain engineering phase and continued during the application engineering phase – In fact, it is recommended to adapt the paradigm to the necessities of each SPL. To the best of our knowledge, there is a lack of systematic guidelines that prevents the practitioners from these peculiarities. Motivated by this, we have formulated a first approach to what could be a flexible – yet still systematic – approach. The approach is based on the principles stated by [23], and completed

with different strategies stated of the SPL testing literature, like mappings and reviews [8,10,28]. Our goal is to begin a discussion around the model with the community in order to enrich it and provide a solid version that can be used as a reference for new and senior SPL developers.

The remainder of this paper is structured as follows: Section 2 presents the information regarding the software developed for this article. Section 3 presents background information on different SPL theoretical testing approaches. Our proposed approach is described in Section 4 and detailed in Section 5. Finally in Section 6 we present concluding remarks and lessons learned.

2 Running example

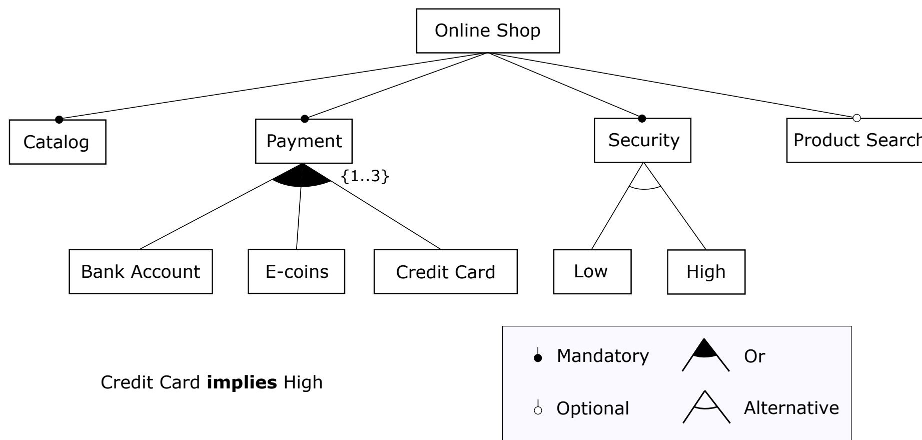

Fig. 1. Online shop feature model [29].

Figure 1 shows the feature model of the online shop example that we will use throughout this paper. It is a configurable online shop system with different capabilities which are the system's variability points, as proposed in [29]. One of the most common methods for modelling variability in industry consist in using feature models [6], in which variability points are mapped into features and then represented in a hierarchical diagram that depicts the relationships between features. The feature model from Figure 1 shows that all online shops must have a catalog listing all the available products, a set of payment methods, and a security level. Furthermore, an online shop can optionally have a search feature which allows users to find products more easily. Further down the hierarchy, we can see that there are three possible payment methods, at least one of which needs to be selected: bank account, e-coins, and credit card. Note that the *{1..3}* cardinality annotation is redundant in this case, since the *or* parent-child relationship requires that at least one or more sub-features are selected. Finally,

the security level of the online shop must be either high or low, since the *alternative* parent-child relationship mandates that exactly one of the sub-features has to be selected.

In addition to the parental relationships between features, feature models may also have additional cross–tree constraints, which are propositional formulas that further reduce the amount of valid configurations. In our example, the "*Credit Card* ***implies*** *High*" constraint makes the feature *High* to be mandatory when the feature *Credit Card* is selected. Taking all of this into account, we can derive a total of 20 valid online shop variants. For more information about feature modeling theory, refer to [4,5].

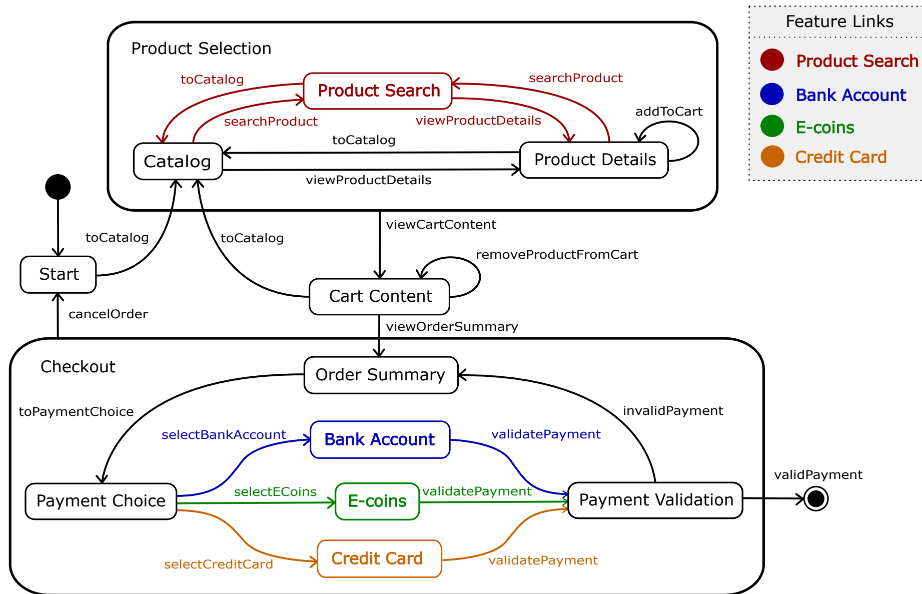

Fig. 2. Online shop 150% model.

One way of modeling the behavior of a software system is by employing a state machine model. This model could also be used to generate test cases if we were using Model-Based Testing (MBT) techniques. In the case of SPLs, the so-called 150% model can be built, which is a domain engineering asset that represents the behavior of the whole product line. 150% models integrate all the variability, i.e., the variability related to the whole product line into one single model [2]. When a specific product variant is selected, the variability of the 150% model is bound, forming the 100% model (i.e., the model specific to that configuration) [2]. This approach is not only considered for models but can also be used for generic code. Figure 2 shows the 150% state machine model of the online shop example, where the links to the features from the feature model have been represented with colors. Note that in many cases the 150% model

itself may not be a valid product variant, since it is not always valid to select all the features on a single product.

Analogously to the 150% state machine model shown, other domain engineering assets can be generated, such as use case diagrams, class diagrams, and even the software source code itself. These assets should also be linked to the feature model so that they can be automatically reused for different products. In order to demonstrate how to manage an SPL project, we have implemented a simple, Java based version of our online shops example using FeatureIDE [27], a tool which can be used to develop SPLs using the feature-oriented software development paradigm.

3 Top-down vs Bottom-up approach

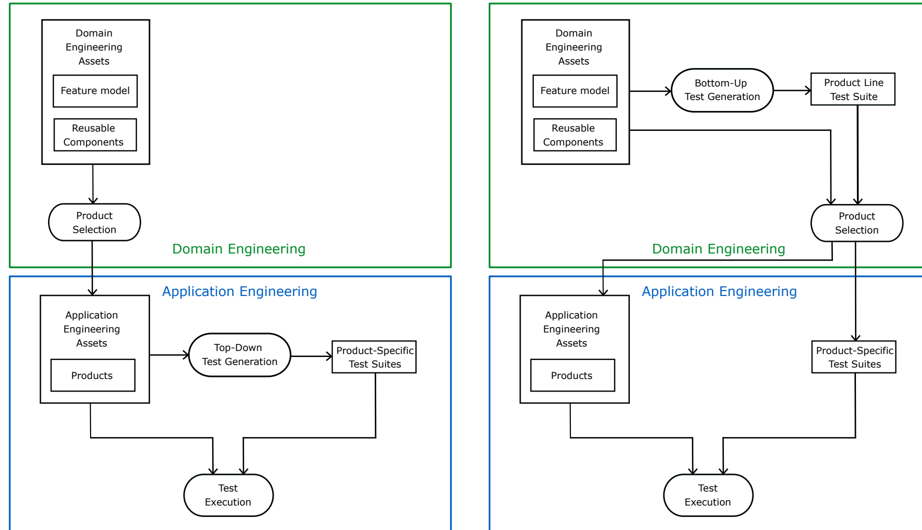

Fig. 3. SPL testing Top-Down approach.

Fig. 4. SPL testing Bottom-Up approach.

There are two main approaches at the SPL testing: (1) the top-down approach and (2) the bottom-up approach [29], which are also referred to as product-centered and product line-centered respectively in some publications [19]. On the one hand, the top-down approach consists in selecting and generating the desired product variants first, and then generating a test cases for each derived product individually. On the other hand, the bottom-up approach consists in generating a database of generic test cases for the whole SPL based on the domain engineering assets, such as the 150% model. Later, variability of these test cases is bound to test individual product variants. Figures 3 and 4 show an overview of these approaches.

toCatalog, searchProduct, viewProductDetails, searchProduct, toCatalog, viewProductDetails, addToCart, toCatalog, viewCartContent, removeProductFromCart, toCatalog, viewCartContent, viewOrderSummary, cancelOrder, toCatalog, viewCartContent, viewOrderSummary, toPaymentChoice, (selectBankAccount OR selectECoins OR selectCreditCard), validatePayment, validPayment

Example 1. Bottom-up test case for online shops.

We define a test case as a specification of inputs used for software testing. In this paper, they will consist of sequences of events that trigger transitions in our example state machine model.

As an example of the bottom-up approach, if we wanted to generate a product line test suite to obtain a high transition coverage for the online shops example, we could come up with the test case shown in Example 1, where the colors represent the same feature links as in the Figure 2 model.

toCatalog, searchProduct, viewProductDetails, searchProduct, toCatalog, viewProductDetails, addToCart, toCatalog, viewCartContent, removeProductFromCart, toCatalog, viewCartContent, viewOrderSummary, cancelOrder, toCatalog, viewCartContent, viewOrderSummary, toPaymentChoice, selectECoins, validatePayment, validPayment

Example 2. Derived bottom-up test case for online shops.

After product selection, we can derive this test case into product-specific test cases for every product, allowing us to reuse it. Note that this test case only covers one of the payment methods, even if multiple are selected. If, for instance, we selected a product with $\{ProductSearch, ECoins, LowSecurity\}$, its derived product-specific test case would be Example 2.

As for the top-down approach, the product-specific test suite will be generated after selecting the product variants, so we can just leverage existing software testing techniques and tools.

It has been observed by some authors that the bottom-up approach seems to scale better than the top-down approach in terms of test execution cost (considering the total test case count, number of steps and number of configurations) [19]. Furthermore, considering that the top-down approach is similar to the traditional testing system, this paper will be more focused in the bottom-up approach.

4 Proposed Process Overview

This proposal is based on the strategy *Design test assets for reuse* [8]. This means that test plans and test cases are created as soon as possible, usually in domain engineering. Nevertheless, application engineering tests are still needed,

so it is also important to encourage the reuse of those product-specific test cases defined in application engineering from one to another product.

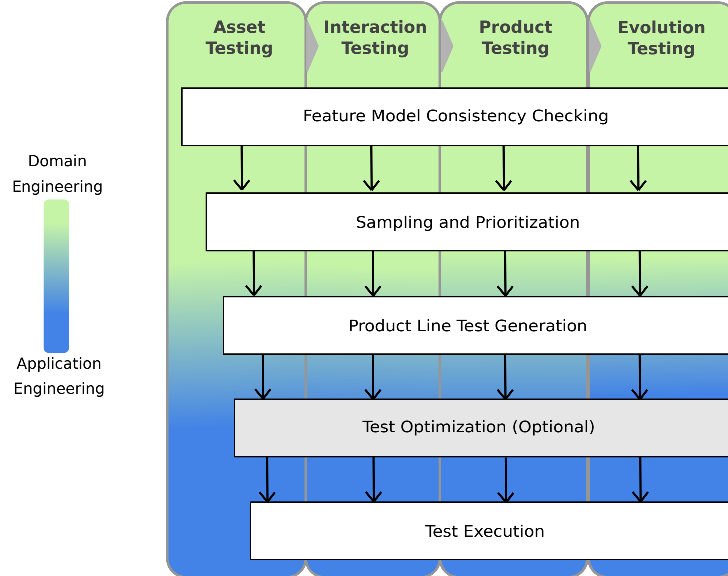

Fig. 5. SPL testing process timeline proposal.

Figure 5 shows an overview of the activities involved in the testing process of an SPL and their mapping into domain or application engineering. It is possible to divide the SPL testing workflow in four different procedures: *Asset Testing*, *Interaction Testing*, *Product Testing* and *Evolution Testing*. These procedures are sorted in chronological order in Figure 5, but in high variability environments it is usually impossible to avoid mixing them. One example is how in [23] it is described how it is not necessary to develop and test every asset before starting to test a product by applying testing prioritization.

There are SPL testing processes that are affected and modified by all of the aforementioned procedures. They are represented by white boxes in Figure 5, and the arrows represent the dependency between them. In order to achieve an optimal test execution firstly we need to ensure that every feature selected (or added to the SPL due to latter product needs) is consistent with the original feature model – *Feature model consistency checking* [1]. After that, prioritization of tests should be established for every single procedure. Most times, and because of the variability of the SPL, sampling techniques have to be applied in order to achieve good coverage of interaction testing between assets. This is referred to as *Sampling and Prioritization*.

Once we have a clear view of which interactions and processes should be prioritized on the SPL testing, a generic test plan is built. Named as *Product Line Test Generation*, this process is heavily influenced by both domain and application engineering, and depends on the already tested assets. It is essential to unify and reuse test assets (test cases, test scenarios and test results) as much as possible from one product to another. To achieve this, product-specific created test assets are stored in the generic SPL testing plan as they appear, as this has been proven to impact the effort reduction [8].

Finally, *Test Optimization* can optionally be performed on application engineering to achieve an optimal *Test Execution* for every procedure.

5 Procedures

In this section, every sub-process from Figure 5 is define. Furthermore, some glimpses about different techniques every process are given.

5.1 Asset Testing

Assets are defined as artifacts built for the development of different products of the same SPL [3]. In the literature they are also referred to as *domain artifacts* or *core assets*. These assets can be tested independently through unit testing making use of traditional software techniques [8]. Results from these tests are valuable for every product derived from a SPL. This process is usually performed in domain engineering, although there are some assets that cannot be tested until there is a product [23].

Depending on the perspective of the SPL development, different approaches can be made. According to [23], it is highly recommended to test *commonalities* first. Commonalities are those assets considered core in the SPL and that will be present in every product. This will be useful in case that a *reference application* [21] wants to be used at the interaction testing procedure. Once commonalities are tested, the testing of variability-affected assets can begin. In order to achieve an optimal coverage of interactions, there can be a prioritization of which variable assets need to be tested first. This will allow interaction testing to start earlier in the testing process.

Example 3 shows a test case for the ProductDetails class in our online shops example, which checks the presence or absence of the ProductSearch option in the user menu. Even if ProductDetails is a core asset that is always present, this particular test case cannot be executed until a product is selected because the transition to ProductSearch may or may not exist.

5.2 Interaction Testing

Asset testing is not enough for achieving a high quality SPL. In SPLs, interaction between assets causes failures, bugs and inconsistencies that can only be

```
1    public void testSearchTransition() {
2      // Select menu option
3      input.println("2"); // 2. BACK TO CATALOG
4      // Run ProductDetails
5      productDetails.run();
6      List<String> lines = output.lines().collect();
7      // Check output
8      boolean searchProductPresent = false;
9      for(String line : lines) {
10       if(line.equals("3. SEARCH PRODUCT")) {
11         assertFalse(searchProductPresent);
12         searchProduct = true;
13       }
14     }
15     // #if ProductSearch
16     assertTrue(searchProductPresent);
17     // #else
18     assertFalse(searchProductPresent);
19     // #endif
20   }
```

Example 3. Asset test for the ProductDetails class.

detected when certain feature combinations are present. For this reason, interaction testing is used when testing SPLs. Variability is controlled on this stage of SPL testing [7], and results will be valuable for every different products derived from the SPL. Interaction testing includes integration testing – which belongs to domain engineering [8] – and also binding testing [23], which belongs to application engineering.

In high variability scenarios it is impossible to achieve a full coverage of every interaction between assets. This is caused by the number of products an SPL can have, that grows exponentially as the number of features increases. Therefore, sampling is necessary to test as many different interactions as possible while avoiding exhaustive testing [28]. The idea behind sampling is to derive a subset of all possible products that collectively cover the behavior of the SPL and reveal most of the faults by only them [28].

To sample a product subset from the entire SPL, several approaches have been proposed. Varshosaz et al., proposed a taxonomy to classify these approaches [28]. This taxonomy included (1) input data to the sampling approach (e.g., feature model), (2) type of algorithm used for sampling products (e.g., from simple greedy-based algorithm to more sophisticated population-based algorithms) and (3) type of coverage employed (e.g., feature-interaction coverage). Out of the scope of this paper, the classification also included the evaluation technique and the type of application of the approach, among which most of them were focused on testing.

The input data covers the artifacts that the sampling approaches consider for generating products. All the approaches considered feature models as input, something that is quite sensible since this is the technique that is applied most in industry [6]. However, according to [28], some approaches also combined feature models with expert knowledge or implementation artifacts. The algorithm is referred to the process that is behind the product sampling to search for relevant products to test, which uses a specific type of coverage to guide this search. While most of the approaches are based on Greedy algorithms (e.g., ICPL algorithm [16]), in the last few years, more sophisticated techniques, such as population-based algorithms (e.g., GAs) have been proposed [14]. Most of these algorithms are guided by certain criterion, which is mostly driven by feature-wise and pair-wise coverage (i.e., they follow the principle of combinatorial interaction testing). This is a way of measuring coverage in SPL sampling, which aims at, for every combination of t assets, to cover all interactions at least once, which requires 2^t test cases for each combination.

Combinatorial testing has been proven to produce good results in empirical studies [9,25], mainly because points of interaction between software artifacts have been proven [18] to be key sources of errors. This type of sampling is usually automatically performed by an algorithm which aims to minimize the amount of samples selected to achieve the desired t-wise coverage, reducing the testing cost while maximizing the test value.

- $P_1 = \{ECoins, LowSecurity\}$
- $P_2 = \{BankAccount, CreditCard, HighSecurity, ProductSearch\}$
- $P_3 = \{BankAccount, ECoins, HighSecurity, ProductSearch\}$
- $P_4 = \{CreditCard, HighSecurity\}$
- $P_5 = \{BankAccount, LowSecurity\}$
- $P_6 = \{ECoins, LowSecurity, ProductSearch\}$
- $P_7 = \{ECoins, CreditCard, HighSecurity\}$

Example 4. Pairwise product selection for online shops.

For our motivating example, seven out of twenty possible products have been sampled in order to obtain the highest pairwise feature-interaction coverage. By employing a feature model modeled in FeatureIDE as input, the ICPL [16] and as driving algorithm and pairwise as coverage criterion, the sampled products listed in Example 4 have been derived.

5.3 Product Testing

A good practice is to generate a general SPL testing plan and test suite for product testing at a domain engineering level, just taking into account interaction and asset testing results. Nevertheless, there is still a need of performing a general testing for every product derived from an SPL before delivering it to

the customers after that. There are certain tasks – like acceptance testing or non-functional testing [8] – that can only be fully tested when the product is derived.

In this step, we are capable of generating a test-plan by consuming the results from previous stages. For example, taking into account the features included in the product, or guarantying that certain non-functional requirements meet customers requirements.

> *toCatalog, viewCartContent, viewOrderSummary, toPaymentChoice, selectBankAccount, validatePayment, invalidPayment, toPaymentChoice, selectCreditCard, validatePayment, validPayment*

Example 5. Product-specific test case for online shops.

In Example 1, we proposed a reusable test case which can achieve a high behavioral coverage for all cases. Nevertheless, that test case will only cover one of the payment methods even if multiple are selected, and it also does not cover the *invalidPayment* transition. Even though it is possible to make a reusable test case with full coverage for our simple running example, it is easier to extend the reusable test case with a product-specific test case once variability is resolved. The test case in Example 5 could be generated for the product $\{BankAccount, CreditCard, HighSecurity, ProductSearch\}$ in order to compliment the test case in Example 1. Note that this test case focuses on the Checkout states, since the other states are already fully covered.

5.4 Evolution Testing

Evolution testing [20] is performed at more mature stages of the SPL life-cycle to ensure that the behavior of the products and the whole SPL system remains valid after modifications (such as bug fixes or functionality extensions). Even though this paper is centered in testing on the developing stage of an SPL, it needs to be noted that evolution testing should be performed in order to ensure that the behavior of the system remains valid and consistent with the feature model [13].

6 Conclusions

Although the existing literature already states that variability blurs the difference between domain and application testing processes [23,15], a testing paradigm where this situation is clearly specified hadn't been proposed yet. This article proposes a generalist and non-technique-focused SPL testing approach to make an advance reuse of test assets (Figure 5). In order to justify it, a brief compendium of actual SPL testing techniques and their flaws have been exposed. One of the benefits from this is that SPL developers do not need to change their

own procedures when developing, and that there is no tool dependency. Furthermore, this approach can be partially and conveniently applied depending on the SPL complexity.

We felt that a conceptual approach was necessary in order to guide new SPL practitioners into the SPL testing paradigm, but it is also certain that this approach is highly conceptual. Some highlights of future work would be the development of a more technique-specific and low-level procedure, as well as the evaluation of this approach on high scale projects.

Acknowledgements

This work has been partially funded by the EU FEDER program, the MINECO project OPHELIA (RTI2018-101204-B-C22); the Juan de la Cierva postdoctoral program; the TASOVA network (MCIU-AEI TIN2017-90644-REDT); the Junta de Andalucia METAMORFOSIS project and the Basque Government via TEKINTZE project.

Material

The prototype of the SPL we prsented as running example can be found at `https://github.com/jonayerdi/OnlineShops`

References

1. Mauricio Alférez, Roberto E Lopez-Herrejon, Ana Moreira, Vasco Amaral, and Alexander Egyed. Supporting consistency checking between features and software product line use scenarios. In *International Conference on Software Reuse*, pages 20–35. Springer, 2011.
2. Aitor Arrieta, Goiuria Sagardui, Leire Etxeberria, and Justyna Zander. Automatic generation of test system instances for configurable cyber-physical systems. *Software Quality Journal*, 25(3):1041–1083, 2017.
3. Felix Bachmann and Paul Clements. Variability in software product lines. Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.
4. Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.
5. David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
6. Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.
7. Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, ROSATEA '06, pages 53–63, New York, NY, USA, 2006. ACM.

8. Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5):407 – 423, 2011. Special Section on Best Papers from XP2010.
9. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *PROC. INTL. CONF. ON SOFTWARE ENGINEERING (ICSE '99)*, pages 285–294, 1999.
10. Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes*, 37(6):1, November 2012.
11. José A. Galindo, David Benavides, and Sergio Segura. Debian packages repositories as software product line models. towards automated analysis. In *ACoTA*, pages 29–34, 2010.
12. José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. Automated analysis of feature models: Quo vadis? *Computing*, Aug 2018.
13. Jianmei Guo, Yinglin Wang, Pablo Trinidad, and David Benavides. Consistency maintenance for evolving feature models. *Expert Syst. Appl.*, 39:4987–4998, 04 2012.
14. Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-objective test generation for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 62–71, New York, NY, USA, 2013. ACM.
15. L. Jin-hua, L. Qiong, and L. Jing. The w-model for testing software product lines. In *2008 International Symposium on Computer Science and Computational Technology*, volume 1, pages 690–693, Dec 2008.
16. Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55. ACM, 2012.
17. Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.
18. D.R. Kuhn, D.R. Wallace, and Jr. Gallo, A.M. Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, 30(6):418–421, 2004.
19. Hartmut Lackner, Martin Thomas, Florian Wartenberg, and Stephan Weißleder. Model-based test design of product lines: Raising test design to the product line level. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 51–60. IEEE, 2014.
20. Miguel A. Laguna and Yania Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming*, 78(8):1010 – 1034, 2013. Special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages.
21. Beatriz Pérez Lamancha, Macario Polo Usaola, and Mario Piattini Velthius. Software product line testing. *A Systematic Review. ICSOFT (1)*, pages 23–30, 2009.
22. Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media, 2005.

23. Klaus Pohl and Andreas Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, December 2006.
24. Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. In *VaMoS*, pages 45–51, 2010.
25. Ben Smith and Martin S. Feather. Challenges and methods in testing the remote agent planner. In *In Proc. 5th Int.nl Conf. on Artificial Intelligence Planning and Scheduling (AIPS*, pages 254–263, 2000.
26. Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 254–264. IEEE, 2009.
27. Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.
28. Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A classification of product sampling for software product lines. In *Proceeedings of the 22nd International Conference on Systems and Software Product Line-Volume 1*, pages 1–13. ACM, 2018.
29. Stephan Weißleder and Hartmut Lackner. Top-down and bottom-up approach for model-based testing of product lines. *Electronic Proceedings in Theoretical Computer Science*, 111, 03 2013.