Automated Variability Analysis and Testing of an E-Commerce Site. An Experience Report

Sergio Segura, Ana B. Sánchez and Antonio Ruiz-Cortés
Department of Computer Languages and Systems
University of Seville, Spain
{sergiosegura,anabsanchez,aruiz}@us.es

ABSTRACT

In this paper, we report on our experience on the development of *La Hilandera*, an e-commerce site selling haberdashery products and craft supplies in Europe. The store has a huge input space where customers can place almost three millions of different orders which made testing an ex-tremely difficult task. To address the challenge, we explored the applicability of some of the practices for variability management in software product lines. First, we used a feature model to represent the store input space which provided us with a variability view easy to understand, share and discuss with all the stakeholders. Second, we used techniques for the automated analysis of feature models for the detection and repair of inconsistent and missing configuration settings. Finally, we used test selection and prioritization techniques for the generation of a manageable and effective set of test cases. Our findings, summarized in a set of lessons learnt, suggest that variability techniques could successfully address many of the challenges found when developing e-commerce sites.

Keywords

Variability; automated testing; experience report; feature modelling; e-commerce

1. INTRODUCTION

Variability is a pervasive feature of modern software applications that determines their ability to be configured and customized. Software applications exposing a high–degree of variability are usually referred to as *variability–intensive systems*. Operating systems as Linux [2], development tools as Eclipse [16] or even cloud applications as the Amazon elastic compute service [10] have been reported as examples of variability–intensive systems.

Software Product Line (SPL) engineering focuses on the development of families of software products by systematically managing variability. Feature models are the *de–facto* standard for variability modelling in SPLs. A *feature model* is a visual and compact representation of all the configurations (a.k.a. products) of an SPL in terms of features and relations among them [17]. Feature models are also used to enable the interactive configuration of variability–intensive systems by selecting and deselecting features [1, 24].

The automated analysis of feature models deals with the computer–aided extraction of information from feature models. Catalogues with up to 30 different analysis operations on feature models have been reported [1]. Among others, these operations allow to know what is the number of configurations represented by a feature model or whether a feature model contains any *dead* features, i.e. features that cannot be part of any configuration. Also, a recent trend on the automated analysis of feature models focus on reducing the potentially huge testing space of SPLs. These operations take a feature model as input and return a manageable subset of configurations to be tested using test case selection [7, 21, 23, 27, 31] and test case prioritization techniques [6, 9, 12, 30].

Online shopping systems (a.k.a e–commerce sites or online stores) have also been proposed as a good example of variability–intensive systems [4, 19]. A standard e–commerce site might be composed of tons of modules providing functionality such as shipping management, product rating, user wish list, product search, etc. In turn, each module commonly has numerous setting options leading to millions of potential configurations of the store. E–commerce sites are usually developed on top of off–the–shelf e–commerce solutions such as Magento [22] or Prestashop [29]. Prestashop is an open source e–commerce platform written in PHP for the development of online shopping systems. It offers over 3,500 modules and visual templates powering more than 150,000 online stores worldwide.

In this paper, we report our experience in the development of La Hilandera e–commerce site[1]. La Hilandera (hereinafter the client) is a small Spanish company selling haberdashery products and craft materials. Their main target is the Spanish market although they also sell their products in English to another nine European countries. Founded in 2013, the company currently has three employees managing a company blog, several social profiles and both a physical and online store. Some of their direct competitors are I do proyect [14], TricotPlus [33] and we are knitters [34].

[1]`http://www.lahilandera.com/en`

The online store was developed using Prestashop v1.5 in three main steps. First, the requirements were gathered from the client. Second, a visual template and a set of Prestashop modules providing the required functionality were collected, installed and slightly customized. In total, the functionality of the store is currently supported by 54 Prestashop modules. Finally, the numerous setting options of each module were adjusted to meet the requirements. This involved setting hundreds of parameters related to shipping, payment, product catalogue, languages, taxes, cache, images, Search–Engine Optimization (SEO) and many others. Also, a number of configuration constraints were defined, e.g. certain carriers only ship packages of certain weight to certain zones. The final configuration of Prestashop determines the input space of the store, this is, the input parameters (e.g. payment method), their possible values (e.g. bank wire, credit card and PayPal) and the constraints among them (e.g. free shipping is restricted to Spain).

1.1 Challenges

During the development of the store we found a number of challenges listed below.

1. *Configuration view.* Configuration options and constraints in Prestashop are scattered through a plethora of menus and wizards. The lack of a common view of the configuration hindered the communication among the stakeholders and the traceability between the requirements and the configuration settings.

2. *Detection of inconsistent configuration settings.* The numerous configuration options of Prestashop made easy to make mistakes resulting in inconsistent or contradictory configurations settings. For instance, we could set carriers that can never be selected or weight ranges served by no carriers. We found no modules supporting the automated detection and repair of inconsistent configuration values in Prestashop.

3. *Detection of missing configuration settings.* Checking if a certain input combination was allowed by the current configuration of the store required either laborious tests or diving into a number of menus. For instance, we may want to check whether the current configuration allows a certain carrier to ship orders under 0.5kg to Luxembourg. We found no automated support for this.

4. *Test case selection.* Tests were manually performed introducing GUI input actions (e.g. placing orders) and searching for unexpected behaviour such as wrong or missing configuration settings, wrong translations, CSS incompatibilities, PDF generation problems, etc. However, the high number of input combinations made exhaustive testing impractical. We found no automated means for the selection of a manageable set of test cases with a good coverage of the input space.

5. *Test case prioritization.* Our testing resources were limited and so we needed to run first those test cases that could reveal bugs in the most frequent settings. For instance, our client mainly expected orders under 1Kg from Spain paid with credit card. Thus, we needed to design test cases giving priority to those values while still keeping a reasonable coverage of the rest of the input space. Again, we did not find any Prestashop module supporting test case prioritization.

Previous challenges were manually faced during the development of the online store. This was a time–consuming and unreliable process where the detection of faults mainly depended on our ability to select effective test cases. After a thorough search in the literature and the Web[2], we found that this was the usual process in Prestashop were developers have no automated support for testing their stores.

1.2 Experience report

Based on our research background, we hypothesized that previous challenges could be automatically addressed using variability management techniques. To test our hypothesis, we revisited the main milestones of the development exploring the applicability of well–established techniques for variability management in SPLs. In this paper, we report such experience. To keep our work manageable, we focused only on the functionality related to order placing since that was the most challenging part of the development in terms of testing.

First, we designed a feature model including the main input parameters of the store and their possible values according to the current configuration. Roughly speaking, the model represents all the distinct orders that can be placed in the store. This model provided us with a visual and common view of the store configuration and its input space easy to understand, share and discuss with all the stakeholders (challenge 1).

Next, we automatically analysed the feature model to extract information from the store. Among other data, we found that we were facing an input space of three millions of potential input combinations. Also, we automatically detected several configuration problems, i.e. input values that could never be selected (challenge 2). Besides this, configuration tools made straightforward to check the validity of a given input combination by simply selecting and deselecting features. We found that this was a simple and highly automated mechanism to search for missing configuration settings (challenge 3).

Next, we automatically analysed the feature model to select a manageable and effective subset of input combinations to be tested, i.e. test cases. In particular, the test space was reduced from three millions to 91 test cases including all the valid pairs of inputs, i.e. pairwise suite. This means a reduction of more than 99.9% in the number of test cases (challenge 4). Then, we applied a weighted prioritization algorithm reordering the test cases according to the expected frequency of use of each input value. This was a natural approach to accelerate the detection of critical faults affecting to the core functionality of the store (challenge 5).

As a result of our experience, we present a set of lessons learnt for researchers and practitioners in the field of software variability. We also share or vision of how the proposed variability techniques could be integrated into a hypothetical Prestashop module supporting the generalizability of the approach. Although further research is needed, our findings suggest that SPL techniques could successfully address many of the variability challenges found during the development of an e–commerce site. To the best of our knowledge, this

[2]This included inquiries in Prestashop forums.

is one of the few applications putting into practice the techniques for the automated analysis of feature models in a real scenario.

The rest of the paper is structured as follows. In Section 2, feature models and SPL testing are introduced. The feature model used to represent the variability of the store input space is presented in Section 3. Section 4 details how the analysis of feature models could contribute to the early detection and repair of faults caused by wrong configuration settings. Section 5 illustrates how the SPL techniques for test case selection and prioritization could be used to support the automated generation of test cases. A set of lessons learnt are presented in Section 6. We share our vision of how the approach could be generalized in Section 7. The related work is reviewed in Section 8. Finally, we summarize our conclusions in Section 9.

2. BACKGROUND

2.1 Feature models and their analyses

Feature models (FMs) are commonly used as a compact representation of all the valid configurations of an SPL [17]. An FM is visually represented as a tree–like structure in which nodes represent features and connections illustrate the relationships between them. These relationships constrain the way in which features can be combined to form valid configurations, a.k.a. *products*. For example, the FM in Fig. 1(a) (taken from [1]) illustrates how features are used to specify and build software for mobile phones. The software loaded in the phone is determined by the features that it supports. The hierarchical relationships among features can be divided into:

- **Mandatory**. If a feature has a mandatory relationship with its parent feature, it must be included in all the configurations in which its parent feature appears. In Fig. 1(a), all mobile phones must provide support for `Calls`.
- **Optional**. If a feature has an optional relationship with its parent feature, it can be optionally included in all the configurations including its parent feature. For instance, `GPS` is defined as an optional feature of mobile phones in the example.
- **Set relationship**. A set relationship relates a parent feature with a set of child features using group cardinalities. A *group cardinality* is an interval such as $\langle n..m \rangle$ limiting the number of different child features that can be present in a configuration in which their parent feature appears. The symbol '*' is often used to denote the maximum number of children in the set. In Fig. 1(a), software for mobile phones can provide support for `Camera`, `MP3` or both of them in the same configuration.

In addition to hierarchical relationships, FMs can also contain *Cross–Tree Constraints (CTCs)* between features typically represented using first order logic formulas. For instance, suppose the following CTC is the context of the example, `Camera` $\rightarrow$ `Colour` $\vee$ `HD` i.e. mobile phones including a camera require a colour or a high definition screen.

FMs can be automatically analysed to extract information from them. Catalogues with up to 30 analysis operations on FMs have been published [1]. Typical analysis operations allow us to know whether an FM is consistent (i.e. it represents at least one configuration), what is the number of configurations represented by an FM or whether an FM contains any errors. Also, FMs are commonly used to derive configurations by selecting or deselecting features according to user's preferences. This process is supported by so–called *product configurators*. These tools usually support decision propagation such that whenever a decision is taken (i.e. a feature is selected or deselected) the configuration engine propagates those decisions to enforce their consistency automatically selecting or deselecting the remaining features. In the example, selecting `Basic` screen would implies the automated deselection of the features `Colour` and `HD` (due to the set relationship) and the feature `Camera` (due to the CTC defined above).

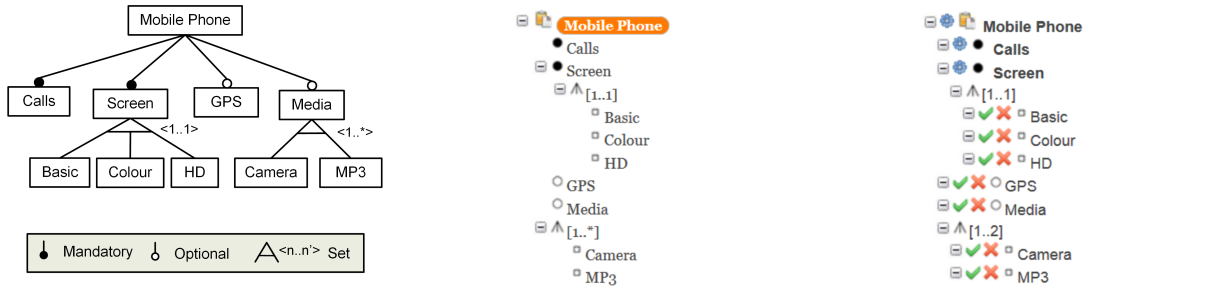
The automated management of FMs is supported by a number of commercial and open source tools including the *FaMa* framework [8], and SPLOT [24]. In this paper, we used the SPLOT online tool suite which includes an FM editor (Fig 1(b)), an FM analysis engine and a product configurator (Fig 1(c)).

2.2 SPL testing

A *SPL test case* can be defined as a configuration of the product line to be tested [27], i.e. a set of features. Ideally, all the configurations of the SPL should be tested although that is often impractical due to the potentially huge number of configurations under test. For instance, As an example, Debian Wheezy [5] has more than 37,000 packages that can be combined (with restrictions) to form millions of different configurations of the operating system which makes exhaustive testing infeasible. To address this challenge, many approaches follow a model–based strategy in which the SPL feature model is used as input to derive a subset configurations to be tested, i.e. a SPL test suite. In particular, two main strategies have been adopted: test case selection and test case prioritization.

Test case selection [21, 23, 27, 31] aims at reducing the test space by selecting an effective and manageable subset of configurations to be tested. Most common test selection approaches are those based on combinatorial testing [18, 21, 23, 27]. In these approaches test cases are selected in a way that guarantees that all combinations of t features are tested, this is called t–wise testing [27]. One of the best–known variants of combinatorial testing is the 2–wise (or pairwise) testing approach [13, 21, 23, 27]. This proposal generates all the possible combinations of pairs of features based on the observation that the most of faults originate from a single feature or by the interaction of two features [27].

Test case prioritization [6, 12, 30] schedules test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal, e.g. detecting faults as soon as possible. There are many possible goals of prioritization [30]. For example, testers may wish to order their test cases in order to achieve code coverage at the fastest rate possible or increase the rate of fault detection of test cases. Also, test cases may be prioritized according to the user preferences or non–functional properties by assigning weights to the features [9, 16].



(a) Graphical view (b) SPLOT tree view (c) SPLOT Product Configurator

Figure 1: A sample feature model

3. VARIABILITY MODELLING

Fig. 2 depicts a feature model representing the variability in the input space of La Hilandera online store. The model is shown using the SPLOT tree view due to space constraints. It was manually designed to reflect the current configuration of the store. Roughly speaking, the model represents all the possible orders that can be placed in the system. Non–leaf features represent the input parameters introduced by the user when placing an order, e.g. payment method. Analogously, child features represent the possible input values of the parameter represented by their parent feature, e.g. bank wire, credit card, PayPal. Both, input parameters and values were modelled according to the current configuration settings of the e–shop. When the input space of a parameter was not restricted to a specific set of values, we divided the domain into so–called equivalence partitions [3, 25] where the store is expected to behave in the same way, e.g. configurable vs. non–configurable products. A *valid configuration* is composed of a set of features (i.e. input parameters and their respective values) that do not violate any of the constraints of the model. Each valid configuration of the model represents an input combination with distinct implications on how the orders are placed and processed.

The model tree is divided into five main branches resembling the five–step checkout process in Prestashop 1.5, i.e. Login→Shopping cart→Address→Delivery→Payment. Additionally, an extra branch is used to model the `Language` settings. In particular, the user interface is available in two languages, `English` and `Spanish`. Each language has an obvious impact in the text translations but also in some of the banners and menus displayed. For instance, the links to the blog of the company (in Spanish) are omitted in the English version of the store.

The `Login` branch refers to the status of the user who place the order, `Guest` or `Customer`. A guest is an anonymous user that must provide contact and shipping information for each order while a customer is a registered user of the site. In turn, `Registered` customers can log in with their user name and password before placing an order or, if they are not customers, they can create a `New` account which requires a different authentication flow.

The `Shopping cart` branch groups the mains choices regarding the type, price and weight of the purchased products. Regarding the product type, products can be classified as `Configurable` and `Non-configurable` products. Configurable products are derived from a so–called *base product* by assigning different values to its configuration attributes such as colour or size. We distinguish between configurable products with the `Same price as the base product` and those with `Different price than the base product`. Regarding the product price, it can be classified as `Normal price`, `Special price`, `Price discount` and `Quantity discount`. Special prices are those marked as such in the administration panel and displayed with a specific format in the store. Discounts are temporal price reductions created using the Prestashop *price rules*. Quantity discounts establish price reductions for buying a number of items greater than a certain value. We distinguish between product purchases where the quantity discount is `Granted` and those where it is `Not Granted`. The cart `Weight range` affect to the selection of the shipping carrier and shipping rates. As requested by the client, we created five weight ranges, namely: `From 0 to 2Kg`, `From 0 to 5Kg`, `From 0 to 10Kg`, `From 2 to 10Kg` and `From 5 to 10Kg`. Note that some of the weight ranges overlap. This was necessary in Prestashop where each carrier must be associated to one or more specific weight ranges. Finally, the `Total to pay` can be `Greater than the FREE_SHIPPING_PRICE` (currently 90€), getting a free shipping, or `Less than the FREE_SHIPPING_PRICE`.

The `Address` branch includes two main choices, the `Destination` country and the selection of the `Billing and shipping` addresses. The destination country has obvious implications in the shipping rates applied. It is noteworthy that the shipping configuration of some European countries was similar and so we grouped them under the `Europe` feature to simplify CTCs. We may remark, however, that the selection of each country still has different implications in how the orders are processed, e.g. postal code format. Beside this, the selection of a single address or two different addresses for billing and shipping has also an impact on the views and logic code exercised in the online store e.g. new address form.

The `Delivery` branch includes the choices regarding the `Carrier` as well as some other options as `Gift wrapping` and `Customer note`. Each choice have effects on the processing of the order. For instance, the selected carrier[3] determines the shipping rate as well as the URL for package tracking sent to the user. The free shipping feature has an obvious impact on the shipping rate that should be reflected in the

[3]Carrier names are omitted for confidentiality reasons.

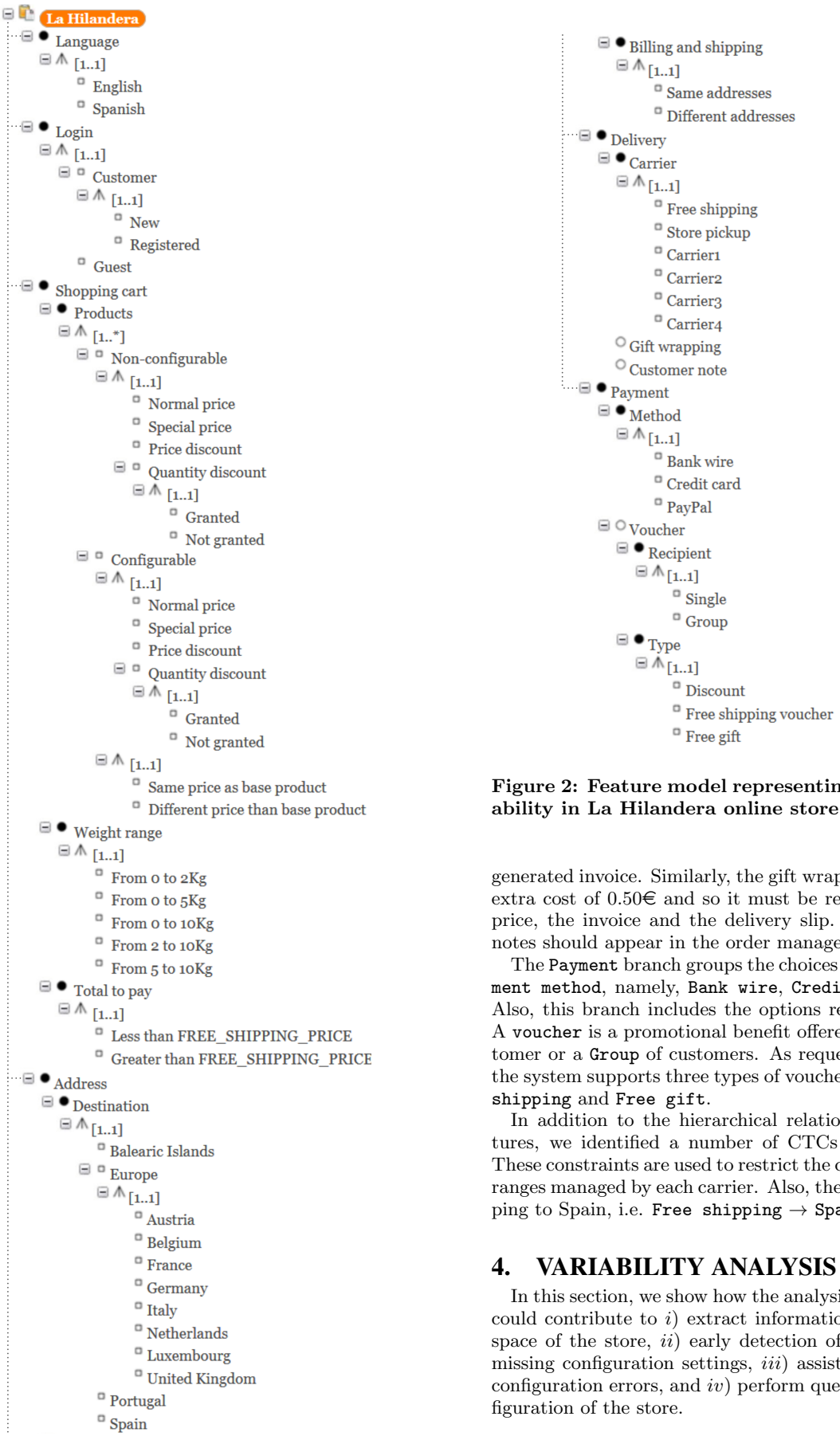


Figure 2: Feature model representing the input variability in La Hilandera online store

generated invoice. Similarly, the gift wrapping option has an extra cost of 0.50€ and so it must be reflected in the final price, the invoice and the delivery slip. Finally, customer notes should appear in the order management dashboard.

The `Payment` branch groups the choices related to the `Payment method`, namely, `Bank wire`, `Credit card` or `Paypal`. Also, this branch includes the options related to vouchers. A `voucher` is a promotional benefit offered to a `Single` customer or a `Group` of customers. As requested by the client, the system supports three types of vouchers, `Discount`, `Free shipping` and `Free gift`.

In addition to the hierarchical relationships among features, we identified a number of CTCs listed in Table 1. These constraints are used to restrict the countries and weight ranges managed by each carrier. Also, they restrict free shipping to Spain, i.e. `Free shipping` → `Spain`.

4. VARIABILITY ANALYSIS

In this section, we show how the analysis of feature models could contribute to *i*) extract information about the input space of the store, *ii*) early detection of inconsistencies or missing configuration settings, *iii*) assist the user in fixing configuration errors, and *iv*) perform queries about the configuration of the store.

Balearic Islands → Carrier4
Europe → Carrier3
Portugal → Carrier1
Spain → Carrier2 ∨ Free shipping ∨ Store pickup
Carrier1 → From 0 to 5kg ∨ From 5 to 10kg
Carrier2 → From 0 to 10kg
Carrier3 → From 0 to 5kg ∨ From 5 to 10kg
Carrier4 → From 0 to 2kg ∨ From 2 to 10kg
Free shipping → Greater than FREE SHIPPING PRICE
Free shipping → Spain
Free shipping voucher → Spain

Table 1: Cross–tree constraints

Regarding the extraction of information, Table 2 depicts some of the statistics obtained from the feature model using the SPLOT analysis tool. As illustrated, the model has 76 features which places it among 3% of the largest models currently stored in the SPLOT feature model repository (out of 527 models). Also, the model has 11 extra CTCs and a CTC Ratio (CTCR) of 22%, being 10% the average value of the models in SPLOT. The CTCR is the ratio of the number of features involved in the CTCs to the total number of features in the model [1]. This gives an idea of how constrained is the input space of the store. Most features (57) are in OR/XOR groups representing the possible values of the input parameters. Analogously, parameters are mainly modelled as mandatory (15) and optional features (3). The model represents 2,985,840 valid configurations, i.e. distinct orders.

Features	76
- Mandatory	15
- Optional	3
- Grouped	57
XOR groups	17
OR groups	1
Tree depth	5
CTCs	11
CTC ratio	22%
Configurations	2,985,840

Table 2: Feature model statistics

Among other operations, the automated analysis of feature models enables the detection of inconsistencies. As an example, the manual tests of the store revealed a *dead carrier*, that is, a carrier that could not be selected due to wrong configuration settings. The problem was caused by a change in the original requirements. Initially, the client wanted Carrier3 to ship all orders to European countries (excluding Spain). However, the client later decided that orders to Portugal would be shipped by Carrier1 which offers cheaper rates. Although Carrier1 had been added to the system it had not been associated to Portugal causing the problem. Fig. 3 depicts how the dead feature would have been automatically detected using the SPLOT Feature Model Editor. This could be complemented by tools as FaMa which implements error diagnosis techniques to identify the causes of inconsistencies, so–called *explanations* [1], e.g."*Carrier1 has no destination countries associated*". Thus, we could get information about why the carrier cannot be selected assisting the user on repairing the problem, i.e. defining Portugal as an independent destination and making Carrier1 its default carrier (`Portugal` → `Carrier1`). This would have allowed us to detect and fix the problem during the configuration stage saving costly testing and debugging resources.

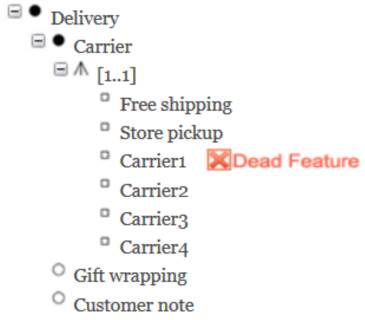


Figure 3: Dead carrier detected during variability analysis

Interactive product configuration may also be handy way to detect missing configuration settings. For instance, during the tests of the store we found that the free shipping vouchers were not restricted to orders in Spain, as requested in the requirements. Fig. 4(a) shows how a product configurator could have been used to detect the problem. As illustrated, once the `Free shipping voucher` is selected, all destinations are still enabled, revealing the bug. Fig. 4(b) shows the configuration view once the missing CTC is added, i.e. `Free shipping voucher` → `Spain`. Note than only Spain remains enabled after selecting the free shipping voucher feature. This approach could be used to perform preliminary tests during the modelling stage by simply selecting and deselecting features instead of diving into tons of menus and wizards.

Finally, product configurators could also be used to perform queries about the current configuration of the store. To that purpose, the user may select or deselect features checking the input combinations allowed by the current configuration settings. For instance, the user could find out if `Carrier1` can ship orders to `Germany` by checking if both features can be selected simultaneously.

5. VARIABILITY TESTING

In this section, we show how the techniques for variability testing in SPLs could contribute to automate the selection and prioritization of test cases for the e–commerce site. We define a *test case* in our domain as a valid configuration of the feature model, i.e. a set of features.

To reduce the test space, we used the SPLCAT tool for pairwise SPL test case selection [15]. This tool takes a feature model as input and returns a pairwise test suite, i.e. a set of configurations containing all the possible pairs of features. Using the feature model in Fig 2 as input, the tool generated a pairwise suite composed of 91 test cases (out of 2,985,840) which means a reduction over 99.9% in the number of configurations under test. More importantly, the pairwise suite is not only manageable but also effective according to the combinatorial testing theory which state that the most faults are triggered by one feature or the interaction between two features [27]. As an example, see below one of the test cases included in the pairwise suite.

(a) Missing configuration constraint

(b) Correct configuration

Figure 4: Using a feature model configurator to detect missing configuration settings

TC: {`Language`, `English`, `Login`, `Customer`, `Registered`, `Shopping cart`, `Products`, `Non-configurable`, `Non-conf Quantity discount`, `Non-conf Not granted`, `Configurable`, `Conf Price discount`, `Different price as base product`, `Weight range`, `From 5 to 10Kg`, `Total to pay`, `More than FREE_SHIPPING_PRICE`, `Address`, `Destination`, `Europe`, `Austria`, `Billing and shipping`, `Different addresses`, `Delivery`, `Carrier`, `Carrier3`, `Gift wrapping`, `Customer note`, `Payment`, `Method`, `Paypal`, `Voucher`, `Recipient`, `Single`, `Type`, `Free gift`}

The test case is composed of the set of features (i.e. input parameters and values) that should be exercised when placing an order in the store under test. Note that presenting a test case as a list of features to be tested would have little or no meaning at all for most Prestashop users. Thus, each leaf feature could be mapped to a sentence in natural language creating step–by–step instructions easy to understand and follow by non–expert users. Not only that, the Prestashop database could be automatically inspected to suggest products that fulfil the criteria defined in the test case, e.g. non–configurable products with price discount under 2Kg. This would make the test case reproducible. Table 3 illustrates a sample user–oriented test case derived from the test case shown above.

Once generated, test cases could be automatically prioritized to meet certain performance goals. To illustrate this, we applied a plain weighted prioritization algorithm. This algorithm receives a test suite where each feature has a weight indicating its importance ranging from 0 (irrelevant) to 1 (critical). The algorithm sorts test cases according to the normalized sum of the weights of its features. According to the expected frequency of use of each feature, we set a weight of 0.9 for features `Guest`, `From 0 to 10Kg`, `Spain` and `Credit card` and a default weight of 0.5 for the rest of features. Then, we ran the algorithm with the suite generated by SPLCAT obtaining a prioritized pairwise suite. As a result, those test cases exercising the high–priority features would be performed first accelerating the detection of critical faults. This would also guarantee that the most relevant test cases have been exercised in the case that the testing resources are exhausted before running the whole suite. Finally, we may remark that the weights could be adjusted and the algorithm run as many times as needed supporting future changes in the client's priorities.

1.	Select English language.
2.	Log-in as a registered user.
3.	Add the following products to the cart: - 5 x Knitting loom. - 1 x Nylon Zippers (Colour: 304, Size: 25cm)
4.	Set Austria as destination country.
5.	Introduce different shipping and billing addresses.
6.	Select Carrier3.
7.	Add a customer note.
8.	Request gift wrapping.
9.	Introduce voucher code for free gift.
10.	Select payment with PayPal.
11.	Confirm purchase.

Table 3: Sample user–oriented test case

6. LESSONS LEARNT

Our experience exploring the applicability of variability analysis and testing techniques to the development of La Hilandera online store can be summarized in the following lessons learnt:

Lesson 1. Feature models are suitable to represent input variability. Feature models are typically used at the problem level to represent requirement variability in a SPL. In this paper, however, we used a feature model at the solution level to represent the input variability of a single store with a specific configuration. Our experience confirms that feature models are expressive enough to capture the variability of the store input space in terms of their input parameters and their possible values and constraints. In this sense, the model can also be regarded as a compact and intuitive view of the configuration settings of the store. More importantly, using a feature model allows using off–the–shelf tools for the automated detection of configuration inconsistencies as well as the generation of test cases.

Lesson 2. Configuration faults are a key target. We distinguish between software and configuration faults. The former are caused by bugs in the code. The latter are caused by wrong configuration settings. Hence, a module could work correctly but still not meet the requirements of the client due to a wrong configuration, e.g. dead carrier. From a Prestashop user perspective, we found that detecting configuration faults is a key challenge. Every single Prestashop store is likely to have different configuration settings which makes configuration faults a widespread problem with no one–for–all solution. Also, the configuration of stores is often performed by final users with no testing knowledge which reinforces the need for automated tool support. To the best of our knowledge, this is the first approach pointing at the problem of configuration faults in e–commerce platforms.

Lesson 3. Feature model analysis for early testing. Our experience suggests that the analysis of feature models is a handy approach for the automated detection of inconsistencies such as contradictory or missing configurations values. Additionally, the available techniques for error diagnosis in feature models could provide helpful information about the sources of the inconsistencies guiding the user toward quick fixes. These analyses would contribute to the early detection (and repair) of configuration faults in e–commerce sites that otherwise would require costly testing and debugging resources.

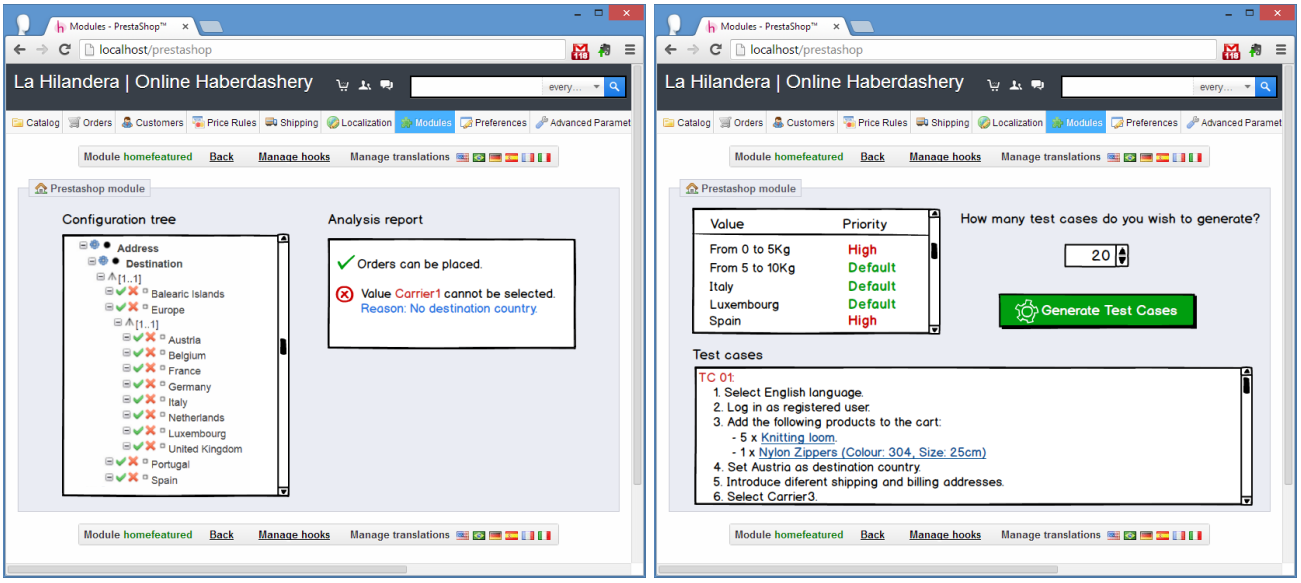
Lesson 4. Priorities emerge naturally. Performing each test case took us between 5 and 15 minutes on average, much more if a failure was revealed and we had to debug the code to fix it. Besides this, we were pressed by the client to release the online store in a specific date previously announced through the social networks. Thus, we naturally felt the need to prioritize our efforts. To that purpose, we intuitively performed first those tests exercising the most common input parameters and values. The goal was to accelerate the detection of those bugs affecting to the core functionality of the store and therefore those likely to generate more losses. Although multiple prioritization criteria could be used, we found that assigning priorities to features according to their expected frequency of use is a quite natural approach for an e–commerce site.

Lesson 5. Integration faults are frequent. This is because new versions of the platform are frequently released creating incompatibilities with previous versions, e.g. modules for Prestashop v1.5 are not backward compatible with Prestashop v1.4. Beside this, the modules and the templates must be compatible with the specific version of Prestashop, but they are often not. In our project, we found that the purchased template had been developed for Prestashop v1.4 and patched to work with Prestashop v1.5 without an exhaustive validation, revealing bugs. Among others, we found that disabling the *product comparison* module triggered a CSS bug that spoiled the appearance of the list of products.

Lesson 6. Four main types of faults. The faults detected in the store can be classified in four main groups: *i*) translation faults, *ii*) CSS faults, *iii*) workflow faults, and *iv*) configuration faults. Translation faults were by far the most common problems. In many cases, the Spanish translation of certain strings was wrong (e.g. spelling mistakes) or simply missing. Also, we found that the Spanish translation of Prestashop uses different ontologies referring to the same term with different synonyms, e.g. shopping cart. CSS faults were mainly related to wrong positioning of the UI elements that required modifying the stylesheet. Workflows faults caused unexpected redirections while browsing the store. Finally, as previously mentioned, configuration faults were caused by wrong settings of the store parameters. Among others, we detected configuration faults related to carriers, shipping and wrapping rates, SEO sitemap files, email notifications and invoice PDF generation. This required several iterations of testing and debugging until gaining confidence in the correctness of the system.

Lesson 7. Full automation is tough. In this paper, we focus on test case generation while the execution of the test cases is out of the scope. Tests were manually performed introducing GUI input actions (e.g. placing orders) and searching for unexpected behaviour such as wrong translations, positioning problems, error messages, wrong navigation workflow, missing images, etc. This involved not only manually checking the front–end user interface but many others artefacts such as the generated PDF documents, order dashboard, customer list, product stock, e-mail alerts, etc. According to our experience, automating the execution of test cases, and in turn the whole testing process, is a daunting challenge. We remark, however, that tools such as PHPUnit [28] and Selenium [32] would be helpful to automated regression tests once the test suite is designed and recorded.

Lesson 8. Poor support for implementing variability. During the development, we received multiple requests from the client to show different information in the Spanish and English versions of the store. This included header and footer links, banners and even products. For instance, the company sells some self–manufactured products (e.g. "do it yourself kits") with Spanish patterns which the client wished to exclude from the English version of the store. Similarly, the client requested that the information about their on–site workshops should be in Spanish only since they are only available for people leaving in the area. Surprisingly, we found no built–in support for this kind of variability in Prestashop. Pressed by time, we were forced to hard code it using basic if/else statements, i.e. `if (lang_iso == 'en') {...} else {...}`



(a) Analysis view

(b) Testing view

Figure 5: User interface of an hypothetical Prestashop module for variability analysis and testing

7. OUR VISION

In this section, we share our vision of how the proposed variability techniques could be integrated into Prestashop generalizing the benefits of our approach. As an example, Figure 5 depicts the user interface prototype of an hypothetical Prestashop module integrating the techniques for variability analysis and testing described in this paper. Note that the target users would be Prestashop users with no expected knowledge about variability or testing. Thus, information should be presented in an intuitive and user–friendly way. Figure 5(a) depicts the *analysis view* where the user would interact with the feature model by selecting or deselecting features. This could be presented either as a feature tree or simply as a list of leaf features with on/off controls. On the right side, a panel would show an analysis report with information about the detected inconsistencies (e.g. dead carrier) as well as their explanations. Figure 5(b) shows the *testing view* where the user should set priorities for each feature (e.g. low, default or high) and the number of test cases to be created. Once generated, test cases would be shown in an additional panel as sequences of steps that the user should perform to detect potential faults in the store. The development of such module would require overcoming several challenges, among others:

1. *Feature model generation.* The feature model should be automatically generated to make the approach generalizable. To that purpose, a feature tree template could be used and populated with the configuration values extracted from the Prestashop database. Such feature tree template should be version–dependent and extensible as new modules are installed.

2. *Inconsistency management.* Researchers have identified different types of inconsistencies in feature models such as dead features, conditionally dead features, false optional features, etc. [1]. Studying which ones of those inconsistencies could appear in Prestashop and how presenting them to the user remains as a challenge.

3. *Generation of user–oriented test cases.* Feature–based test cases should be presented as sequences of steps in natural language easy to follow by Prestashop users. To that purpose, the Prestashop database should be automatically inspected to suggest products that fulfil the criteria defined in the test cases making them reproducible.

8. RELATED WORK

Lau and Czarnecki [19] proposed e–commerce systems as a motivating case study in the context of model–driven SPLs. As a part of their work, they constructed a feature model representing a family of e–commerce systems, so–called *e-shop*. The e-shop feature model has 290 features and represents $4.52 \cdot 10^{49}$ configurations, being the second largest feature model in the SPLOT repository and one of the most referenced in the literature [1]. Their work supports our view on e–commerce sites as motivating variability–intensive systems but with some differences. First, the e-shop model represents a family of e–commerce sites while our model represents the input space of a single store. Second, they focus on variability modelling while we focus on variability analysis and testing. Finally, are more importantly, the e-shop model was constructed from the information gathered in the literature and the Web while our model was derived from a real e–commerce site.

Regarding variability testing, Wang et al. [35] proposed an automated test case selection approach and evaluated it in an industrial case study in Cisco. In [23], the authors reported the results of applying pairwise test case selection in an industrial SPL of video conferencing systems. Nguyen et al. [26] proposed a variability–aware approach to execute test cases in plugin–based web applications and detected two bugs in the popular blogging platform Wordpress. Com-

pared to previous works, we explore the applicability of variability analysis and testing techniques in a novel scenario: a real e–commerce site. Instead of modelling variability at the requirement level, we use a feature model to represent the input space of the store automating the detection of configuration bugs and the generation of test cases.

Equivalence partitioning [3, 25] is a testing technique in which the input domain of the program is divided into partitions (also called *equivalence classes*) in which the program is expected to process the set of data input in an equivalent way. Thus, only an input value of each partition is needed to evaluate the behaviour of the program in that partition. This technique is often combined with combinatorial testing [11] to create input combinations (from different partitions) that are likely to reveal faults while keeping a manageable number of test cases. Our work is mostly inspired by these techniques. The features of the feature model represents the different partitions of the input space of the store while the different relationships among features represents the constraints among the partitions. As reported, this is not only a natural way of representing the input space of an e–commerce site, but it also enables the use of multiple tools for the analysis of variability and combinatorial test case generation.

Yuan et al. [20] presented a survey on web application testing since the origin of the World Wide Web over two decades ago. In their work, authors grouped the testing techniques used in the context of the Web into graph–based testing, model–based testing, mutation testing, search–based testing, random testing, scanning and crawling techniques, concolic testing and user session–based testing. Although these techniques have made significant advances in the detection of faults in web applications, they were not conceived to deal with the specific challenges of variability–intensive systems. Also, these techniques focus on testing web applications as a whole while our approach deals with the specific characteristics of web applications developed on top of open–source platforms as Prestashop.

9. CONCLUSIONS

In this paper, we reported the main challenges found during the development of La Hilandera e–commerce site and explored how they could be addressed using techniques for variability management in SPLs. In particular, we used a feature model to represent all the different orders that can be placed in the store in terms of their input parameters, values and constraints. This allowed us to use off–the–shelf tools to automatically detect configuration bugs and generate effective and manageable test suites. Our findings are summarized as a set of lessons learned for researchers and practitioners in the field of variability. Among others, we identified integration and configuration faults as two key targets where research contribution would be welcome. We also share our vision of how the proposed variability techniques could be integrated into e–commerce platforms as Prestashop generalizing the benefits of our approach to thousands of online stores worldwide. Although further research is needed, this could become one of the few commercial products making profit out of putting into practice the techniques for the automated analysis of feature models. We trust that this work will be used as a motivating and real case study to evaluate variability analysis and testing contributions.

Material

The feature model of the store (in SXFM format), the test suites and the Java implementation of the plain weighted prioritization algorithm are available at `http://www.lsi.us.es/~segura/files/material/ASE14`

Acknowledgments

We would like to thank Dr. Pablo Trinidad whose comments and suggestions helped us to improve the article substantially. This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project TAPAS (TIN2012-32273) and the Andalusian Government projects THEOS (TIC-5906) and COPAS (P12-TIC-1867).

10. REFERENCES

[1] D. Benavides, S. Segura, and A. Ruiz-CortÃ©s. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[2] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, Dec 2013.

[3] L. Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, Inc., Norwood, MA, USA, 2003.

[4] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, GPCE'05, pages 422–437, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] Debian 7.0 wheezy released, May 2013. Accessed November 2013.

[6] X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, and P. Heymans. Towards statistical prioritization for software product lines testing. In *Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, 2014.

[7] F. Ensan, E. Bagheri, and D. Gasevic. Evolutionary search-based test generation for software product line feature models. In *Conference on Advanced Information Systems Engineering (CAiSE'12)*, 2012.

[8] FaMa Tool Suite. `http://www.isa.us.es/fama/`, accessed April 2014.

[9] J. Ferrer, P. KrÃ¼se, F. Chicano, and E. Alba. Evolutionary algorithm for prioritized pairwise test data generation. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, 2012.

[10] J. García-Galán, O. Rana, P. Trinidad, and A. R. Cortés. Migrating to the cloud: a software product line based analysis. In *3rd International Conference on Cloud Computing and Services Science (CLOSER'13)*, 2013.

[11] M. Grindal, J. Offutt, and S. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[12] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the

combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines. Technical report, 2012.

[13] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. 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.

[14] I do proyect. `http://idoproyect.com/`, accessed April 2014.

[15] M. F. Johansen, O. Haugen, and F. Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *MODELS*, 2011.

[16] M. F. Johansen, O. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating better partial covering arrays by modeling weights on sub-product lines. In *International Conference MODELS*, 2012.

[17] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[18] B. P. Lamancha and M. P. Usaola. Testing product generation in software product lines using pairwise for feature coverage. In *International conference on Testing Software and Systems*, 2010.

[19] S. Lau and K. Czarnecki. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, University of Waterloo, Waterloo, 2006.

[20] Y.-F. Li, P. K. Das, and D. L. Dowe. Two decades of web application testing - a survey of recent advances. *Information Systems*, 43:20–54, 2014.

[21] R. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba. Multi-objective optimal test suite computation for software product line pairwise testing. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013.

[22] Magento. `http://magento.com/`, accessed April 2014.

[23] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical pairwise testing for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 227–235, New York, NY, USA, 2013. ACM.

[24] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 761–762, Orlando, Florida, USA, October 2009. ACM.

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

[26] H. Nguyen, C. Kästner, and T. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 6 2014.

[27] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Budry, and Y. le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 2011.

[28] PHPUnit.`http://phpunit.de/`, accessed April 2014.

[29] Prestashop. `http://www.prestashop.com/`, accessed April 2014.

[30] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In *IEEE International Conference on Software Testing, Verification, and Validation*, 2014.

[31] A. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 492–501, Piscataway, NJ, USA, 2013. IEEE Press.

[32] Selenium. `http://docs.seleniumhq.org/`, accessed April 2014.

[33] Tricotplus. `www.tricotplus.com`, accessed April 2014.

[34] We are knitters. `http://www.weareknitters.com`, accessed April 2014.

[35] S. Wang, A. Gotlieb, S. Ali, and M. Liaaen. Automated test case selection using feature model: An industrial case study. In *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 237–253. Springer Berlin Heidelberg, 2013.