Test-based SPL Extraction: An Exploratory Study

Alcemir Santos[1], Felipe Gaia[2], Eduardo Figueiredo[1], Pedro Santos Neto[3], João Araújo[4]

[1] Universidade Federal de Minas Gerais (UFMG) – Minas Gerais – Brazil
[2] Universidade Federal de Uberlândia (UFU) – Minas Gerais – Brazil
[3] Universidade Federal do Piauí (UFPI) – Piauí – Brazil
[4] FCT/CITI, Universidade Nova de Lisboa (UNL) – Lisboa – Portugal

{alcemir, figueiredo}@dcc.ufmg.br, felipegaia@mestrado.ufu.br, pasn@ufpi.edu.br, joao.araujo@fct.unl.pt

ABSTRACT

Many software systems have been developed as single products before Software Product Lines (SPLs) have emerged. Although some promising approaches have been proposed, extracting an SPL from existing software products is still expensive and time consuming. This paper presents an exploratory study that relies on a test-based SPL extraction from systems already developed. We aim to evaluate testing as the main mean to locate feature code and different sorts of existing artifacts to support the test-based location. We conduct two case studies starting from the derivation of the SPL feature model to the feature code location. Our preliminary results indicate (i) good rates of precision for feature seed location, where seed means a small portion of the feature code that allows the identification of the remaining portion, and (ii) good rates of recall for locating the whole feature code.

Categories and Subject Descriptors

D.3.13 [**Reusable Software**]: Domain Engineering.

General Terms

Experimentation.

Keywords

Product line extraction, feature location, software testing.

1. INTRODUCTION

Nowadays, there is an increasing awareness of the benefits in developing reusable software as part of product families. As a result, software industry is guiding its effort to develop their products based on Software Product Line (SPL) concepts [3, 11]. In fact, several large software companies have already adopted SPL engineering to develop their products for more than a decade [11, 14]. However, even before SPL has gained popularity, many software systems had already been developed. This scenario leads to a racing to turn existing software into SPLs. However, the complete extraction of an SPL from existing software systems is expensive and time consuming [4] and error-prone without proper guidance [10]. Furthermore, software systems usually have high coupling [15] and, consequently, code of different features is tangled. Each feature represents an increment in functionality relevant to some stakeholders [20].

SAC'13, March 18-22, 2013, Coimbra, Portugal.

Several research studies have proposed refactoring methods based on static analysis to extract an SPL from existing code [16, 18]. However, code refactoring demands high level of domain knowledge of the target systems [8]. In addition, these methods usually fail because they overlook other available artifacts besides code. In fact, even if variability is implemented at source code level, other enclosed software artifacts, such as requirements documentation and test suites, have also to be taken into account.

Some work [5, 9] has used dynamic methods to support SPL extraction. For example, Eisenbarth and others [5] mapped features to source code when a visual behavior activated by a user was detected. Ghanam and Maurer [9] also mapped executable acceptance test to feature models. Following this pathway, this paper accomplishes an exploratory study on dynamic SPL extraction based on test coverage and existing artifacts of an already developed system. The study includes a four-step environment setting to support SPL extraction. In the first step, a feature model is extracted based on heuristics applied to existing requirements and design documents. In the second step, we performed a mapping from requirements to features. In the third step, the requirements to feature mapping is expanded to also indicate test suites that exercise specific requirements (and consequently features). Finally, in the fourth step our study accomplishes an analysis of novel strategies to locate and annotate the source code that implements each variable feature of the SPL. An innovative flavor of these strategies is the use of integration tests to support the feature location.

We decided to reuse software testing in this study because other studies previously relied on testing reuse [13]. In addition, software tests are usually available on software systems. There are different classifications of software testing. Tests can be classified by their target: one module (unit), several grouped modules (integration), and the entire system (system) [1]. In this paper, we use integration tests because they relates different modules of the system that often implement different SPL features.

The exploratory study was conducted with two applications: a small implementation of a Web-based library management system and an interactive Web store system. The preliminary assessment indicated promising results. In terms of feature seed location, precision values were high, indicating that seeds could be located using testing. A seed means a small portion of the feature code that allows the identification of the remaining portion [21]. In terms of the whole feature code location, where feature code means all the lines of code that belongs to a feature [21], recall values were high. This result indicates that the whole feature code can also be found. However, we had high hate for false positives. Therefore, we observed that dynamic feature code location should be carefully used for this purpose.

This paper is organized as follows. Section 2 presents the study settings, including research questions, target systems, and the evaluation procedure. Section 3 presents an environment setting to support the data collection for the test-based SPL extraction evaluation presented in Section 4. Section 5 discusses some related work. Finally, Section 6 draws some conclusions and points out directions for future work.

2. STUDY SETTINGS

2.1 Research Questions

Our main goal is to evaluate the ability of integration testing to support the SPL extraction. To achieve this goal, the study aims to answer the following questions:

RQ1. Is integration testing effective to locate a feature seed?

RQ2. Is integration testing able to find the feature code?

We analyze how much feature code is discovered by testing. Moreover, we aim to analyze how testing coverage behaves when looking for different kinds of features. For example, we investigate if there are some differences in locating the feature code of the more general features against feature more specifics.

2.2 Target Systems

Our analysis embraced two Web-based systems of different domains. Both systems are written in Java and provide the artifacts needed to perform our study. We spent 2-3 weeks to perform each case study.

The first target system used in this study is a library management system of approximately 1 KLOC, called *JBook*. Its source code is available [19]. It is used to manage the library on a local Brazilian software company and was developed by their developers. On this system, users may play three different roles: reader, librarian and administrator. While readers and librarians have access restrictions to some functionalities, administrators are allowed to do all actions available in JBook. This system allows users, depending on their roles: (i) to register new publications and its exemplars; (ii) manage the loans, and (iii) notify new acquisitions and return deadlines by sending emails.

The second target system, called *WebStore* [6], has also approximately 1 KLOC and its source code is available. It was designed for academic purpose focusing in the major features of a real web store. These features allows the user to insert products, view them by categories and date, as well as perform a checkout and finish the order with different payment methods, such as DEFAULT PAYMENT, PAYPAL and BANKSLIP.

We choose small systems as case studies due to a manual annotation of the features used in the evaluation. Additionally, the exploratory nature of this work favors small systems instead of bigger open-source software project. All files used are available at the study website [24].

2.3 Reference Feature Code

In order to evaluate our results, we ask the developers of JBook and WebStore (experts on these applications) to annotate their source code. For each system, six of the identified features (Section 3.1) were annotated. The experts used a shadowing technique [7] for annotating all feature code. Six features of JBook were annotated: LOAN, LOAN REQUEST, LOAN REPORT, PUBLICATION, EXEMPLAR and USER. The six features annotated for WebStore were PAYMENT, PAYPAL, BANKSLIP, DISPLAY, CHECKOUT, and CONTENT MANAGEMENT.

2.4 Evaluation Procedure

We define two strategies for feature location, each one with a specific purpose. The first one intends to identify just a seed for the feature and the other focuses on the identification of the entire (or the most part) of the feature code. To implement both strategies we have used set operations applied to the lines of code covered by each test suite. We use the intersection set operation to address the seed identification strategy, since it only requires a small fraction of code. That is, the intersection of lines of code executed by all tests of the same feature. On the other hand, we use the union set operation to identify most of feature code to address the feature code strategy. That is, the union of lines of code executed by all tests of the same feature.

To quantify the result of each strategy we use precision and recall metrics presented below. In the metrics, a *true positive (TP)* happens when the line was executed by a test and was considered relevant. A *true negative (TN)* happens when the line was not executed and the line was not considered relevant. A *false positive (FP)* happens when the line was executed and the line was not considered relevant. And finally, *false negative (FN)* happens when the line was not executed and the line was considered relevant. Finishing this process, we build a spreadsheet with the line accounting for calculating the metric values. Figure 1 illustrates these metrics. The set A represents the executed lines and the set B represents the lines that belong to feature.

Precision: this metric measure the fraction of the retrieved lines considered relevant to our purpose.

Recall: this metric measure the fraction of the lines relevant successfully retrieved.

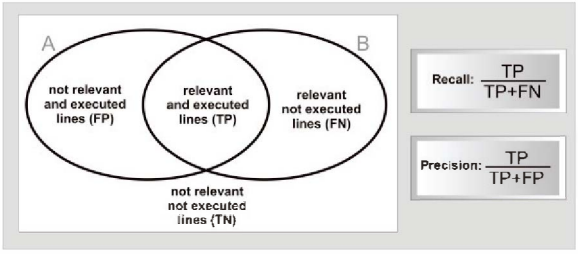

Figure 1. Definition of precision and recall.

The experimental data was collected as follows. After executing the selected test suites and annotating the execution trace of the tests we use the above mentioned feature location strategies to reach a final set of annotated code by feature. Since we have six features of each target system, we got twelve final code sets, six code sets reached by the feature seed strategy addressing JBook and six by the feature code strategy addressing WebStore.

After that, we made a paired comparison between the code sets. Each pair was composed by the reference feature code set and a final code set of covered code identified by our approach. The comparison was made for each feature; leading us to another twelve code sets with lines identified as true positives, true negatives, false positives, and false negatives. We show the analysis of the data collected for both feature seed and code locations strategies in the next sections.

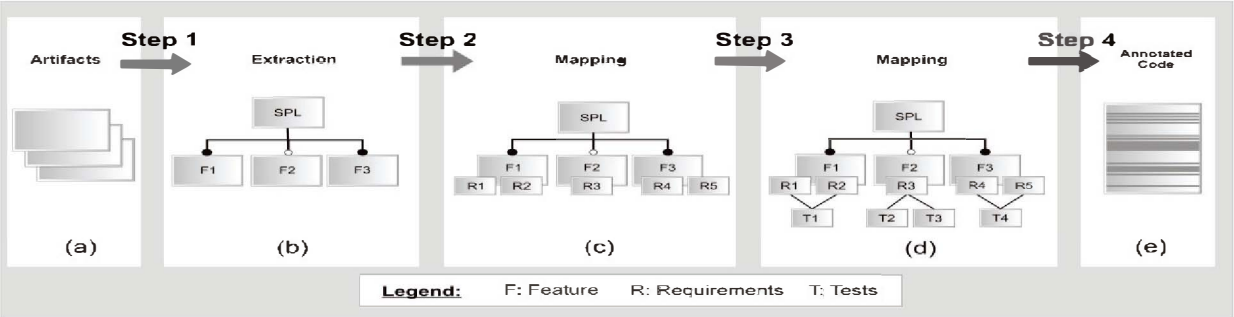

Figure 2. Test-based feature location procedure.

3. TEST-BASED SPL EXTRACTION

We followed the same procedures to evaluate both target systems (Section 2.4). This section presents the environmental settings that use available software artifacts to support test-based feature code location. More precisely, this settings was used not only to extract a partial feature model from requirements and design documents but also to locate and annotate the source code of relevant features. One of the novel characteristics of this approach is the reuse of integration tests for the feature location process.

Figure 2 illustrates the environmental setting in our evaluation. The grey arrows from the left-hand side to the right-hand side indicate the successive steps of the study. The output of each step is on the square identified by (b), (c), (d) and (e). Square (a) illustrates input artifacts of the study. In Step 1, we collected the available artifacts of each system in order to build a partial feature model. Step 2 maps system requirements to features. Square (c) in Figure 2 shows the result of this second step. The requirements are represented by the squares identified by R*j* (*j= 1..5*). Note that a feature can be mapped to more than one requirement. For instance, F1 is mapped to requirements R1 and R2. In Step 3, test suites are associated to features using the mapping of the previous step. Square (d) shows the result of the third step. The tests suites are represented by the squares identified by T*k* (*k= 1..5*). A test suite can be mapped to more than one requirement. For instance, test suite T1 and T4 are mapped to the requirements R1. In the same way, a requirement can also have more than one test suite mapped to. For instance, test suites T2 and T3 are mapped to the requirement R3. Finally, in Step 4, we execute the set of tests attached to a feature in order to locate its code. Once the code was executed, we collect data for further analysis.

Examples of artifacts of systems in square (a) that we use to generate the feature model are requirements, architecture / design models, and tests suites. These artifacts are commonly developed in the software development process and, so, they allow the replication of this study to most of the developed systems, regardless of the specific development process adopted.

3.1 Building the Feature Model

This section describes some heuristics which belong to Step 1 (Figure 2) aiming to support the feature model derivation. Although these heuristics can be adapted and applied to different sorts of existing requirements and design artifacts, we illustrate them by their application to use case diagrams and descriptions of the target systems. The heuristics, named H1-H10, aim to identify the features of the target system. These heuristics, briefly described, are based on previous research work [17]. Table 1 shows some examples of features extracted with the use of the heuristics over the target systems available documentation. With these values in hand, we analyze terms and logically organize them in a feature model. Figure 3 shows the JBook-SPL and WebStore-SPL partial feature model built after applying the heuristics, some feature were hidden due to space constraints.

Table 1. Heuristics and examples of feature extracted.

Heuristics	Feature Examples
H1. Identify the root feature based on use cases;	JBOOK, WEBSTORE.
H2. Identify features based on the use case names;	**JB●●K:** REGISTER USER, UPDATE USER, REGISTER EXEMPLAR, CANCEL LOAN. **WEBST●RE:** INSERTPRODUCT
H3. Identify features that can be grouped based on use case names;	**JB●●K:** REGISTER USER and UPDATE USER can be grouped by PROCESS USER;
H4. Identify features based on the "list of requirements";	**JB●●K:** CHOOSE USER ROLE; ADD, UPDATE.
H5. Identify variability from use case description;	**JB●●K:** NOTIFY USER.
H6. Identify variability for other features of the model;	**JB●●K:** CHOOSE USER ROLE. **WEBST●RE:** BYCATEGORY and WHATISNEW
H7. Identify *xor* alternatives;	**JB●●K:** ADMINISTRATOR, READER and LIBRARIAN.
H8. Identify *or* alternatives;	**JB●●K** - USER OPTION: ADD and UPDATE. **WEBST●RE** - PAYMENT: PAYPAL and BANKSLIP.
H9. Identify requires dependencies relationships in feature model;	**JB●●K:** VALIDATE USER DATA.
H10. Identify excludes dependencies relationships in feature model:	In our case studies we could not identify any example.

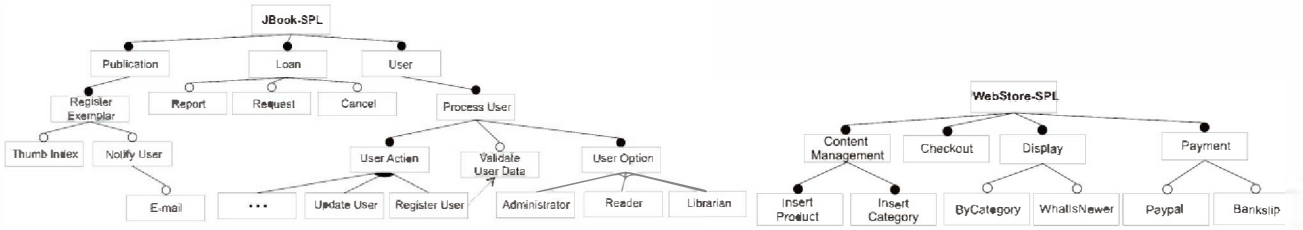

Figure 3. A partial JBook-SPL and WebStore-SPL feature models built after analysis of documentation available.

3.2 Mapping Requirements to Features

The second step of our environment setting was the mapping of requirements to features of the previously generated feature model. Since the use case descriptions define a list of requirements of the system, we can also use them to attach the identified features to requirements. In the JBook, for instance, we have the use cases "User register" and "User password change". These use cases and respective requirements can be easily associated with the feature USER, as illustrated in Figure 4. For simplification purpose, this figure represents only the use cases abstracting their respective requirements. This step later enables the association between features and test suites.

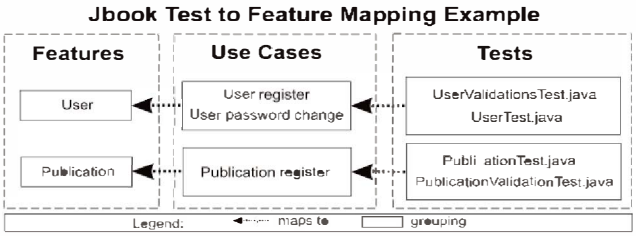

Figure 4. Tests associated to features through use cases.

3.3 Mapping Tests to Feature

Once we know which requirements are assigned to which features, we can also indirectly associate integration tests with features. In other words, the mapping from tests to features can be inferred since (i) tests aim to find defects on the code implementing requirements and, so, they are implicitly or explicitly linked to requirements and (ii) requirements were mapped to the features on the previous step (Section 3.2). In fact, although tests are not explicitly mapped to requirements in developed systems, it is easy to recover this mapping by tests specification. For instance, Figure 4 shows how we can associate tests to use cases. In this case, we hide the packages where tests are located due to size constraints of the figure. The link between tests suites and requirements as well as the link between requirements and features allow us to exercise a feature code by executing the corresponding integration tests.

3.4 Feature Code Annotation

After the third step above, we have test suites associated with the features. Therefore, we can run the integration tests in order to locate the feature code. Figure 5 shows the process executed on this fourth step. First, we select the test suite depending on the feature we intend to address, and then we execute it. The execution of the test suites leads us to a set of executed code shaded by a coverage tool. Finally, we carry on an analysis of the code shaded to finish the location of code that implements a feature. After repeating this process with all the features, we know the feature code that should be used in a software product line.

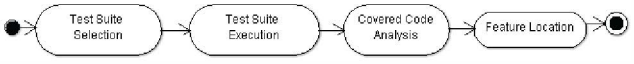

Figure 5. A 4-step process to annotate the feature code.

Algorithm 1 shows a sample of the source code (shaded) that we extracted from JBook after the execution of the `LoanRequestTest` integration tests. The white lines were not executed and might not belong to the feature addressed as well as the dark gray shaded. The light gray shaded piece of code was executed by the test and should belong to the feature. In this case we locate the code of the feature LOAN.

Algorithm 1. An example of JBook test-based shaded code.

```
1  public class Loan implements Serializable{
2     private Long id;
3     private Date requestDate;
4     (…)
5     public Loan(){
6         this.requestDate = new Date();
7     }
8     public Long getLoanId(){
9         return id;
10    }
11 }
```

4. Results and Analysis

4.1 Feature Seed Identification Strategy

This section presents the results when we identify feature seeds by using this strategy on JBook. The feature seed identification strategy consists of using all tests suites related to one feature executed. We then get the seed for a feature by applying the intersection set operation in sets of code annotated by different tests suites. This strategy aims to identify one or more seeds for the feature code and should be used when a seed is enough to find the remaining feature code, i.e., using a different approach.

Figure 6 shows the result for precision in black columns and recall values in white columns for JBook. The LOAN, EXEMPLAR, PUBLICATION and USER features reaches precision rates above 85%. This results indicate that the annotated code is (almost) completely dedicated to the feature implementation; i.e., very few false positive. In the case of the LOAN REQUEST and LOAN REPORT features, the results are not as good as expected. However, we think these values are related to the fact that both features are refinement of the LOAN feature. Therefore, REQUEST-specific tests end up executing LOAN code that does not belong to REQUEST. Similar behavior was observed for LOAN REPORT.

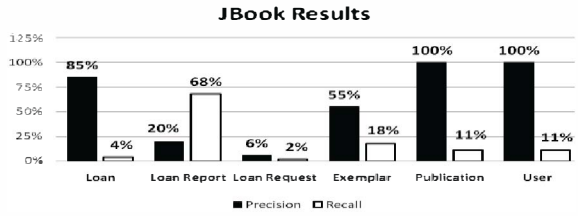

Figure 6. Results for feature seed identification in JBook.

Figure 6 shows that the results for recall are quite low for most features, except for the LOAN REPORT feature. This result is justifiable for this strategy since it discards some correctly recovered lines of code that do not appear in the execution traces of all feature-related tests. One exception is the LOAN REPORT feature that reaches 68% of recall. This result might happen as it is implemented in only two Java classes which turn its code easier to be executed by the tests of the implementing classes.

LOAN REQUEST is a feature with the lower precision and recall values. We deeply investigated this case and found out that two factors impacted on the poor results of LOAN REQUEST. First, very few test cases targets requirements associated with this feature. Therefore, most of the feature code is never executed leading to low recall values. In addition, LOAN REQUEST is a very specific feature implemented by just a few lines of code. Therefore, we observed that test-based feature location cannot easily find this kind of specific and tiny features.

The goal of this strategy is to identify a feature seed. Therefore, in general, a high precision in this case means that most lines of code are relevant as a seed. On the other hand, while precision values are reasonable to the feature seed point of view, low values for recall explains why this strategy is not accurate to find the whole feature code. Therefore, we next discuss the results of a approach which tries to maximize the feature code located.

4.2 Feature Code Identification Strategy

This section presents the results of the feature code identification in WebStore. The feature code identification strategy consists of using all tests suites related to one feature. In this case, we apply the union set operation of all code from different tests of assigned to feature. This strategy aims to identify most of the feature code and, hence, it should be used when a seed is not enough to complete the assigned task. Our goal is to facilitate the feature extracting by indicating a superset of the feature code.

Figure 7 shows the result of precision (black columns) and recall (white columns). Once you implement a test case of a specific method, it needs to execute the code that supports it, even though this code is not directly related to the method under test. Therefore, testing feature code with this strategy leads to many false positives. As a result, we expect the precision values to be low. The higher precision values, PAYMENT and CONTENT MANAGEMENT, are exactly of the bigger features analyzed. Although testing executes much code not belonging to these features, their size contributes to increase the precision values.

Figure 7. Results for feature code identification in WebStore.

Focusing on recall values, all features achieved more than 60%. The worst recall was observed in the case of PAYMENT and CONTENT MANAGEMENT. By analyzing these two features, we realize that these lower recall values are mainly due to the size of the features. In this case, small features are harder to be found by the test-based approach. Similar problem was also observed in other features due to the low number of tests available to locate the feature code. Not always test engineers covers all code with test cases, giving a special attention to portion of the code classified as critics [22, 23].

Once the goal of this strategy is to identify the whole feature code, a higher recall means that it fairly achieved its goal. That is, most lines of code that belongs to the feature were annotated by testing. In fact, we should use this strategy to reduce the space for searching the feature code. For instance, our results indicate that around 70% of the application code is not executed by tests assigned to a feature. Therefore, someone needs to mine a feature code in only 30% which are executed and partially annotated by this strategy. Low precision values indicate that many false positives remain and, therefore, post analysis is needed to complete the extraction of the software product line.

4.3 Discussion

After showing the results, we can answer our research questions. The answer for RQ1 is partially, by using the strategy discussed in Section 4.1. That is, whether integration testing can locate a feature seed if we considered the intersection set of all code executed by different tests of a feature. The answer for RQ2 is also partially. It is possible to extract features using testing under some conditions, such as high test coverage. In addition, it usually requires some additional effort to finish the SPL extraction. The results showed that testing coverage has a great power of feature location with recall values over 60% for all features addressed in both JBook and WebStore. We also achieved good precision rates on feature seed location, with about 60% in average for all features addressed in both case studies. Therefore, we observed that the feature seed identification strategy shows a reliable path to start an SPL extraction process.

4.4 Threats of validity

One limitation of this study is the size of target applications systems. We used two small applications for the study because larger systems could be an impediment for building the feature code reference list used in the data analysis. The manual work to build the reference list of feature code is another limitation of our study. We rely on experts who must understand most of the system code to build the oracle. Even though it was made by experts and later revised by our team, it may contain some assignment errors. Additionally, the integration tests must execute the code related to the analyzed feature or, at least, small pieces of them depending on of the test coverage. JBook has a higher coverage than we usually find on typical systems (around 90%). WebStore has a common coverage scenario of around 50%.

SPL is usually extracted from a set of products in the same domain. In this paper, we took only one product application into account. However, test suites might be available to all different products, which mean that testing might represent a good path on the identification of core assets, as well as, variable features in either scenarios: a single product or multiple products.

5. RELATED WORK

The feature location aims at locating pieces of code that implement a given set of features. Eisenbarth and others [5] presented a semi-automatic technique to rebuild the mapping of feature which has a visual behavior observable when activated by the user. In addition, Antoniol and Guéhéneuc [2] presents an approach to identify features by using static and dynamic data on object-oriented multitask systems. Poshyvanyk [12] made use of two techniques to locate features: scenario-based probabilistic ranking; and an information retrieval based by using latent semantic index. So, they argued a significantly increase in the effectiveness of the location if compared with the use of the two techniques separately. Walkinshaw [18] combined the most important methods to the feature implementation annotations and built a graph with all direct paths between pairs of methods annotated. Then, they added all the methods which could be influenced by it. In the end, they eliminated the irrelevant methods that were added during this process. All these works intend to locate features on the source code to help developers in the maintenance and evolution of legacy software. However, our purpose is to identify the source code that implements a feature with fine granularity in order to separate the feature code and generates different products into an SPL.

Another closer work was done by Ghanam and Maurer [9]. They proposed to attach acceptance tests on the feature model by using executable acceptance tests (EAT). Each EAT were added to the lowest level of the feature model (leaves). Their goal was to track features in code artifacts. The assessment of their work was qualitative. Our work is different because we generated a SPL from legacy software systems instead of tracking the code through the feature model. We decided to use testing on feature location after carrying out a previous work [13]. We also promoted the test reuse and endorsed its importance on software development.

6. CONCLUSION

This paper presents an exploratory study on the use of testing to support the extraction of an SPL from developed software as a single product. We conduct the extraction of the SPL since early stages, when the feature model is created, to the identification of code implementing each variable feature. We use two systems as case studies, called JBook and WebStore. In addition, the study defines novel strategies to locate and annotate the source code that implements each variable feature of the SPL. We show some results related to the feature identification in the case studies in terms of precision and recall.

The results presented in this study were interesting and the test-based technique can be considered on an SPL extraction scenario. Additionally, based on the research questions answered (Section 4.3), we conclude that the test-based technique can reduce the time needed for the SPL extraction by indicating either (i) a good seed or (ii) a superset of the feature code. Besides, much of the software developed nowadays has testing suites and this fact reduces the costs of this approach.

In future work, we intend to automate the mapping process to support the test-based feature location and conduct other studies on test-based SPL extraction. We are also planning further experiments on the relations between features and the issues related to their size. Finally, the number of times each line was executed by some sorts of tests could help us with valuable information and also deserves further investigation.

ACKNOWLEDGMENTS: We would like to acknowledge CNPQ: grants 131788/2011-6 and 485235/2011-0; FAPEMIG: grants APQ-02932-10 and APQ-02376-11; AMPLE project and CITI-PEst-OE/EEI/UI0527/2011.

7. REFERENCES

[1] Bourque *et. al.* 2001. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, USA.

[2] Antoniol, G. and Guéhéneuc, Y. 2005. *Feature identification: A novel approach and a case study*. Proc. of the Int'l Conf. on Software Maint. (ICSM), pp. 357-366.

[3] Chastek, G. J., McGregor, J. D. 2008. *Production Planning in a Software Product Line Organization*. Proc. of the Int'l Software Product Line Conference (SPLC), pp. 369-369.

[4] Couto, M., Valente, M. and Figueiredo, E. 2011. *Extracting Software Product Lines: A Case Study Using Conditional Compilation*. Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR).

[5] Eisenbarth, T., *et. al*. 2003. *Locating features in source code*. IEEE Trans. on Software Eng. (TSE), v.29, pp. 210-224.

[6] Ferreira, G., Gaia, F., Figueiredo, E. and Maia, M. 2011. *On the Use of Feature-Oriented Programming for Evolving Software Product Lines – a Comparative Study*. Proc. of the XV Brazilian Symp. of Progr. Lang. (SBLP), pp. 121-135.

[7] Figueiredo, E. *et. al*. 2008. *Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability*. Proc. of Int'l Conf. on Software Engineering (ICSE), pp. 261-270.

[8] Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Publishing Boston, USA.

[9] Ghanam, Y. and Maurer, F. 2010. *Linking feature models to code artifacts using executable acceptance tests*. Proc. of Int'l Software Product Line Conf. (SPLC), pp. 211-225.

[10] Kästner, C., Dreiling, A. and Ostermann, *K. 2011. Variability mining with LEADT*. Technical report, Philipps University Marburg.

[11] Muhammad A. N., Rick R. and Paul G. 2008. *Agile product line planning: A collaborative approach and a case study*. Journal of Systems and Software (JSS), vol. 81, pp. 868-882.

[12] Poshyvanyk, D. *et. al.* 2007. *Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval*. IEEE Transactions Software Engineering (TSE), vol. 33, pp. 420-432.

[13] Santos, I., Santos, A. and Santos Neto, P. 2011. *Reusing Functional Testing in order to Decrease Performance and Stress Testing Costs*. Proc. of Int'l Conf. on Software Engineering and Knowledge Eng. (SEKE), pp. 1-5.

[14] Sharp, D. 2000. *Reducing Avionics Software Cost Through Component Based Product Line Development*, Proc. of Software Product Line Conference (SPLC).

[15] Taube-Schock, C. *et. al*. 2011. *Can we avoid high coupling?* Proc. of the European Conf. on OOP (ECOOP), pp. 25-29.

[16] Valente, M., Borges, V. and Passos, L. 2012. *A Semi-Automatic Approach for Extracting Software Product Lines*. IEEE Trans. of Software Eng. (TSE), 38(4), pp.737-754.

[17] Varela, P., Araújo, J., Brito, I. and Moreira, A. 2011. *Aspect-oriented analysis for software product lines requirements engineering*. Proc. of ACM Symposium on Applied Computing (SAC), pp. 667-674.

[18] Walkinshaw, N., Roper, M. and Wood, M. 2007. *Feature location and extraction using landmarks and barriers*. Proc. of the Int'l Conf. on Software Maint. (ICSM), pp. 54-63.

[19] JBook. *http://sourceforge.net/projects/jbookweb/*

[20] Kästner, C., Apel, S. and Kuhlemann, M. 2008. *Granularity in software product lines*. Proc. of the Int'l Conf. on Software Engineering (ICSE), pp. 311-320.

[21] Marin, M. Deursen, A. V. and Moonen, L. 2007. *Identifying Crosscutting Concerns Using Fan-In Analysis*. ACM Trans. of Software Eng. Methodologies (TOSEM), 17, article 3.

[22] Tassey, G. 2002. *The economic impacts of inadequate infrastructure for software testing*. Planning Report. National Institute of Standards & Technology (NIST), pp. 2-3.

[23] Binder, R. 1999. *Testing object-oriented Systems – Models, Patterns and Tools.* Addison Wesley Reading.

[24] Test-based SPL Extraction. *http://www.dcc.ufmg.br/~figueiredo/spl/extraction*