Towards Automated Testing and Fixing of Re-engineered Feature Models

Christopher Henard*, Mike Papadakis*, Gilles Perrouin†,◇, Jacques Klein*, and Yves Le Traon*
*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg, Luxembourg
†Precise Research Center In Software Engineering (PReCISE), University of Namur, Namur, Belgium

*{firstname.lastname}@uni.lu; †{firstname.lastname}@fundp.ac.be

***Abstract*—Mass customization of software products requires their efficient tailoring performed through combination of features. Such features and the constraints linking them can be represented by Feature Models (FMs), allowing formal analysis, derivation of specific variants and interactive configuration. Since they are seldom present in existing systems, techniques to re-engineer FMs have been proposed. There are nevertheless error-prone and require human intervention. This paper introduces an automated search-based process to test and fix FMs so that they adequately represent actual products. Preliminary evaluation on the Linux kernel FM exhibit erroneous FM constraints and significant reduction of the inconsistencies.**

***Index Terms*—Feature Model, Testing, Fixing, Search-based**

I. Introduction

Variability-intensive Systems (VIS) form a vast class of software whose role is to support mass-customization of (software) products and adaptation to increasingly more complex situations formed by mixes of versatile environments, challenging user needs and time-to-market constraints. We all use such systems when we buy a product online (there exists more than 800 web-based *configurators* [1]) or configure an operating system kernel such as Linux [2] for a particular need. Variability is key to these systems but difficult to handle given the large number of constraints linking the features. It is also a great source of combinatorial headaches as the number of possible configurations exponentially grows with the number of options offered. Furthermore, variability is rarely treated as a first-class concept, raising issues for VIS analysis, (re-) engineering and quality assurance.

In sharp contrast, Software Product Line Engineering (SPLE) handles variability as a first-class concept through Feature Models (FMs) [3]. In essence, a FM aims at defining legal combinations of features authorized or supported by a system (*configurations*) using hierarchical decomposition and additional constraints. FMs are now equipped with formal semantics [4], automated reasoning operations and benchmarks [5], tools [6] and languages [7]. There are also used to derive products [5], configure them [8] and for automated quality assurance [9]. A FM can also be converted to a boolean formula [10] to be used within SAT solvers [11] for reasoning and analysis. We will use this representation for our reasoning on their correctness. A recent survey on the analysis of FMs and its use in literature can be found in [5].

To bring the benefits of feature modeling to VIS, several re-engineering techniques [2], [12], producing FMs from various systems artifacts have been proposed. These techniques are partly automated and often require human intervention. Such *re-engineered* FMs may thus not be accurate, yielding incorrect analyses decisions about VIS, in turn hampering their correct re-engineering. As an example of such inaccurate re-engineering, our experiments show that none of the 1,000 configurations generated from the Linux kernel FM [2], [13] is consistent with respect to actual kernel configuration rules. This context motivates our two research questions:

[RQ1] *How to detect inconsistencies between the re-engineered FM and its source VIS?* Inconsistencies fall into two categories. On the one hand, system configurations derived from the FM are incorrect with respect to the system. On the other hand, existing valid configurations do not satisfy the FM formula. We refer as *testing* the process of finding these flaws. When detected, dealing with these discrepancies may require an automated correction of the FM.

[RQ2] *How to automatically make a FM consistent with its real system?* Digging manually through thousands of features and dealing with hundred thousands of possibly faulty constraints in a FM is not an option. Thus, one must devise automated ways to correct inconsistencies in the FM so that it reflects its system. The process of correcting a FM is referred to as *fixing*. Current re-engineering approaches either do not validate the re-engineered FMs or use simulation of the configuration process [2]. This practice, as our experiment shows, is insufficient to detect all the problems of the FM.

In this paper, we propose an automated approach to both test and fix re-engineered FMs. It relies on a continuous loop where FMs are iteratively tested and fixed. This loop forms a search process that gradually improves (fixes) the FMs. The search is guided by the number of inconsistencies found during a continuous testing process. Early results on the Linux kernel FM shows that more than 50% of the problems encountered in the re-engineered FM can be eliminated.

The remainder of this paper is organized as follows: Section II presents the proposed approach. Section III reports on the empirical evaluation. Finally, Section IV discusses related work before Section V concludes the paper.

◇FNRS Postdoctoral Researcher.

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
New Ideas and Emerging Results

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

II. Test-and-Fix Loop

Our approach involves two entities: the system and the re-engineered FM. To find and correct the problems of the FM, such as erroneous constraints, this approach requires two steps. The first one aims at testing the FM to highlight the problems. The second step uses feedback information from the testing process to fix the FM. The repetition of these two steps form a test-and-fix loop illustrated by Figure 1.

A. Testing the Feature Model

Testing a FM consists of two parts: 1) the evaluation of the FM consistency using valid configurations of the system[1] and 2) the evaluation of configurations generated from the FM with respect to the system. This process is depicted by the upper box of Figure 1.

1) Evaluating the Consistency of the FM with Respect to Valid Configurations of the System: This evaluation goes from the system to the FM. To this end, we assume the existence (or the possibility to obtain by some means) of working and actual configurations of the system. The FM is evaluated over these configurations to find existing constraints in the FM which are not compatible with the existing configurations. This first step gives feedback information on existing wrong constraints (EWC) in the FM. These EWC of the FM, as long as the existing system configurations that fail (SCF) to be validated through the feature model, are returned.

2) Evaluating Configurations Generated from the Feature Model with Respect to the System: This evaluation goes from the FM to the system. Valid configurations of the FM[2] are randomly generated using a SAT solver [14]. These configurations are then evaluated on the system side. To this end, the tester decides whether the configurations are valid with respect to the system and provides feedback when configurations fail. We will consider that generally, the tester defines an oracle which uses abstract rules to decide upon the validity of configurations. The oracle depends on the system: for instance, it can use execution information or compilation result, and represents the task usually done by the tester. The oracle rules that fail (ORF) as long as the generated configurations that fail (GCF) according to the oracle are returned.

B. Fixing the Feature Model

This second step aims at fixing the FM based on the feedback information collecting during the testing part. This step is represented by the lower box of Figure 1.

In the following, a generic FM of n features and m clauses or constraints is considered, with the formula of the FM in conjunctive normal form (CNF). Thus, each constraint C is a disjunction of $k \leq m$ literals, where a given literal represents a selected or unselected feature of the FM. The formula of the FM is of the form $\bigwedge_{i=1}^{n} C_i$.

To fix the FM, three operations are considered:

[1]A valid configuration refers to a working configuration of the system.

[2]Here, a valid configuration is a configuration that satisfies the FM formula.

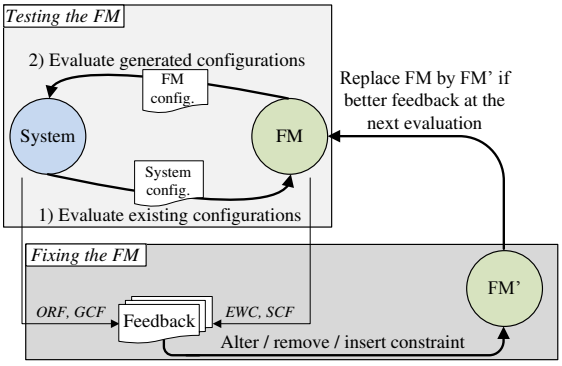

Fig. 1. Test-and-Fix Loop. ORF are oracles rules that are violated on the system side, GCF are the configurations generated from the FM that fail to be validated on the system side, EWC are existing wrong constraints in the FM and SCF are system configurations that do not satisfy the FM,

- *Altering* an existing constraint of the FM. Using the EWC, a constraint of the FM is selected based on a fitness proportionate selection and a randomly selected literal of the selected constraint is negated,
- *Removing* an existing constraint of the FM. Using the EWC, a constraint selected based on a fitness proportionate selection is removed,
- *Inserting* a constraint in the FM. Using the ORF, a constraint is added to the FM.

Performing one of this operation depends on a probability. After having performed one of this operation, an updated FM, FM', is produced.

C. Continuous Improvement

Basically, the global approach consists in the repetition of the testing and fixing steps. To this end, the problem is formulated as a search-based one and a hill climbing technique [15] is used. The approach works as follows. From a given FM, the testing step provides feedback information and the fixing part produces a modified FM FM'. Then, to decide which FM to keep, i.e. the original or the fixed one, the fitness of the original FM and the fitness of the fixed FM' are compared. This comparison is performed using the four feedback information of the testing step:

- s_1: the number of EWC or $\#EWC$,
- s_2: the number of SCF, or $\#SCF$,
- s_3: the number of ORF, or $\#ORF$,
- s_4: the number of GCF, or $\#GCF$.

Let us consider as $s_1, ..., s_4$ the feedback information of an FM and as $s'_1, ..., s'_4$ the feedback information of the fixed FM FM'. The updated FM FM' will replace the original FM if and only if a better fitness is observed for FM'. A better fitness for FM' occurs if the following condition is satisfied:

$$[(\sum_{i=1}^{4} s'_i < \sum_{i=1}^{4} s_i) \wedge (s'_1 \leq s_1 \wedge s'_2 \leq s_2 \wedge s'_3 \leq s_3 \wedge s'_4 \leq s_4)] \vee (s'_2 < s_2 \wedge s'_4 \leq s_4) \vee (s'_2 \leq s_2 \wedge s'_4 < s_4).$$

This condition allows ensuring that a decrease in one of the four feedback information does not engender any negative impact on the others and keeping the focus on reducing the

configurations that fail. Finally, after having replaced or not FM by FM', the process is repeated from step (1). It should be noted that the loop is general and independent of both the way FMs are represented and the use of SAT solvers.

III. Preliminary Evaluation

To evaluate the proposed approach, we consider the Linux kernel 2.6.28.6 FM [2], [13].

A. The Linux Kernel

The Linux kernel is an operating system written in C with about 6,000 features. The re-engineered FM of the Linux kernel contains about 200,000 constraints. In the context of this study, an oracle is needed on the system side to decide whether a given configuration derived from the FM is valid or not. In the context of this study, we use the *make* tool as the oracle. Alternatively, the user could decide himself (play the role of the oracle) about the validity of the configurations. *Make* is a tool that assist the compilation process of system sources. To this end, *make* check rules which are specified by dependencies between the features in *Kconfig* files. These files are placed in the source code directories of the system. We parsed these files to extract these dependencies and to transform them into CNF constraints. It represents around 8,000 constraints. Thus, for a given configuration generated from the FM, it is checked whether this configuration satisfy or not the oracle rules. If yes, the configuration is considered as a valid configuration of the system. Otherwise, it is considered as invalid. It is noted that the use of *make* as an oracle to test is specific to this study. The test-and-fix loop does not generally depend on this oracle.

B. Evaluation of the Re-engineered Linux Kernel FM

The re-engineered FM contains several problems that have been found while performing the testing process. Recall that the testing process allows evaluating the FM through the EWC, SCF, GCF and ORF feedback information.

First, problems occur when evaluating the re-engineered FM formula with respect to valid configurations of the system. An alternative option provided by *make* is the generation of valid configurations of the system. The re-engineered FM has been evaluated over 1,000 working system configurations produced by *make*. By evaluating the FM through these working configurations of the system, we found that 50 constraints in the FM were not satisfied and none of these 1,000 configurations were able to satisfy the FM. In addition, major issues were highlighted such as mandatory features in the FM which never appear in any valid configuration of the system.

Second, configurations generated from the FM do not satisfy the constraints checked by the *make* tool (the oracle rules). We found that for, any 1,000 configurations generated from the FM, more than 28% of these constraints were not satisfied and that all these 1,000 configurations generated from the FM were invalid for the system. These problems are summarized in the column "Re-engineered FM" of Table I. The existence of these problems motivate the proposed approach.

C. Improving the FM

We executed our approach on the re-engineered FM. For all the testing and fixing steps, we used 3 valid configurations of the system. We could have use more valid configurations, but in a realistic situation, only a small number of working configurations should exist. These configurations were randomly generated using *make* and used for all the repetitions of the test-and-fix approach. For the generated configurations, 6 were generated at each repetition of the approach. The probabilities to execute one of the three operations were assigned as follows: 0.5 for the alteration, 0.4 for the insertion and 0.1 for the removal.

Using the testing process, we evaluated the fixed FM at different level of repetitions, as shown in Table I. The feedback information were obtained using the same 1,000 valid system configurations as those used for the evaluation of the re-engineered FM and using 1,000 configurations generated from the FM.

1) RQ1: The evaluation of the EWC, SCF, GCF and ORF of the re-engineered FM emphasizes the inconsistencies between this FM and the system. We believe that these simple metrics characterize inconsistencies that may exist between VIS, FMs representing their variability and oracles checking the legality of VIS configurations. Yet, more detailed inconsistency types may be needed to improve user feedback and drive the fixing process.

2) RQ2: The proposed approach uses both existing valid configurations of the system and configurations generated from the FM. By using only 3 valid configurations of the system, the proposed approach allows reducing wrong constraints in the FM while making the FM satisfiable towards existing configurations of the system. After 5,000 repetitions of the process, the proposed approach dropped system configurations that fail from 1,000 to 455, and divided by more than the half the violated rules of the system.

TABLE I
Evolution of the FM problems over the repetitions (Testing Feedback). EWC are existing wrong constraints in the FM, SCF are system configurations that do not satisfy the FM, ORF are oracles rules that are violated on the system side and GCF are the configurations generated from the FM that fail to be validated on the system side.

	Re-engineered FM	Fixed FM at 2,000 runs	Fixed FM at 3,000 runs	Fixed FM at 4,000 runs	Fixed FM at 5,000 runs
$\#EWC$	50	46	43	41	39
$\#SCF$	1,000	885	556	498	455
$\#ORF$	2,468	1,646	1,395	1,236	1,084
$\#GCF$	1,000	1,000	1,000	1,000	1,000

IV. Related Work

Efforts have been made to re-engineer FMs. She *et al.* introduced procedures [2] to recover constructs such as mandatory features or implies edges to build graphs and provided ranking heuristics to allow the modeler identifying the features hierarchy. In the same context, an approach that builds FMs from the feature sets describing the system variants based on Evolutionary Algorithms has been proposed [12]. Acher *et al.* [16] focused on maintaining a link with the products while reverse engineering FMs. They also took into account the architect's knowledge to build FMs consistent with the software architecture. The method presented in this paper complements these approaches by introducing a way of validating the FM they provide. In addition, our technique goes a step further by automatically fixing the inconsistencies it identifies.

To debug configurations, Hubaux *et al.* [17] proposed a fix-generation approach, called *range fix* to prevent wrong configurations. Their strategy uses constraints solving to propose a list of valid assignments for enabling features. This is done using an underlying model which contains the system constraints. In [18], Tartler *et al.* presented an approach to find and fix defects contained in implementations of configurations of large-scale systems. It is a diagnostic tool which aims at fixing the code. In the same lines, Segura *et al.* [19] presented a framework for benchmarking and testing on the analysis of FMs. This approach is able to automatically detect faults in the analysis tools that operates on a FM. Our approach operates on the FM itself and aims at testing whether its constraints are consistent, and if they are not, to fix them.

Search-based techniques have also been used to perform automatic program improvement. In [20], Langdon *et al.* proposed an approach to improve the lines of code of a system. Based on a fitness function and population of patches, they evolved the original code into a faster version while keeping the semantic of the code at least unchanged or better. Similarly, in [21], Le Goues *et al.* presented a scalable genetic cloud computing oriented technique to repair erroneous programs.

V. Conclusion and Future Work

Fixing a re-engineered FM to make it in consistent with its corresponding system is not an easy task. It requires a lot of efforts to first check existing constraints and then correct them. In practice, it represents a manual work difficult to realize.

In this paper, a test-and-fix loop to automatically improve re-engineered FMs was presented. This loop is implemented using a search-based technique, since the exploration space makes impossible the application of other non-scalable approaches. The proposed approach tries to make the FM conform to the real system. The novelty of this approach is that it achieves to effectively automate the identification and correction of FMs inconsistencies. It is the first approach, to the authors knowledge, that actually employs and checks actual configurations on a real system while most re-engineering techniques do not face the generation of real system configurations. The preliminary study conducted on the Linux kernel FM provides promising results as it allows reducing the problems observed in the FM, thus making the it reflect the real system.

Finally, there is room for improvement. We will first investigate inconsistency types to exhibit fine-grained ones to guide more effectively the fixing process. We will then explore our testing-and-fix approach in the context of VIS evolution.

Acknowledgment

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, under the MITER project C10/IS/783852.

References

[1] http://www.configurator-database.com, 2011.
[2] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE*, 2011, pp. 461–470.
[3] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Technical Report CMU/SEI-90-TR-21, Tech. Rep. CMU/SEI-90-TR-21, 1990.
[4] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux, "Feature diagrams: A survey and a formal semantics," in *RE*, 2006, pp. 136–145.
[5] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, pp. 615–636, 2010.
[6] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "Featureide: A tool framework for feature-oriented software development," in *ICSE*, 2009, pp. 611–614.
[7] D. S. Batory, "Feature Models, Grammars, and Propositional Formulas." in *SPLC*. Springer, 2005, pp. 7–20.
[8] E. K. Abbasi, A. Hubaux, and P. Heymans, "A toolset for feature-based configuration workflows," in *SPLC*, 2011, pp. 65–69.
[9] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. L. Traon, "Pairwise testing for software product lines: comparison of two approaches," *Software Quality Journal*, vol. 20, no. 3-4, pp. 605–643, 2012.
[10] M. Mendonça, A. Wasowski, and K. Czarnecki, "Sat-based analysis of feature models is easy," in *SPLC*, 2009, pp. 231–240.
[11] D. Le Berre and A. Parrain, "The sat4j library, release 2.2, system description," *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 7, pp. 59–64, 2010.
[12] R. E. Lopez-Herrejon, J. Galindo, D. Benavides, S. Segura, and A. Egyed, "Reverse engineering feature models with evolutionary algorithms: An exploratory study," in *SSBSE*, vol. 7515, 2012, pp. 168–182.
[13] "Tools for analyzing variability in the linux kernel," http://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas.
[14] 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," *CoRR*, vol. abs/1211.5451, 2012. [Online]. Available: http://arxiv.org/abs/1211.5451
[15] S. J. Russell, P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards, *Artificial intelligence: a modern approach*, 1996.
[16] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Reverse engineering architectural feature models," in *ECSA*, 2011, pp. 220–235.
[17] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *ICSE*, 2012, pp. 58–68.
[18] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann, "Revealing and repairing configuration inconsistencies in large-scale system software," *International Journal on Software Tools for Technology Transfer (STTT)*, pp. 1–21, 2012.
[19] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés, "Betty: benchmarking and testing on the automated analysis of feature models," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, 2012, pp. 63–71.
[20] W. B. Langdon and M. Harman, "Genetically improving 50000 lines of C++," Department of Computer Science, University College London, Research Note RN/12/09, 19 Sep. 2012.
[21] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012, pp. 3–13.