PLEDGE: A Product Line Editor and Test Generation Tool

Christopher Henard
SnT, University of Luxembourg
Luxembourg, Luxembourg
christopher.henard@uni.lu

Mike Papadakis
SnT, University of Luxembourg
Luxembourg, Luxembourg
michail.papadakis@uni.lu

Gilles Perrouin*
PReCISE, University of Namur
Namur, Belgium
gilles.perrouin@fundp.ac.be

Jacques Klein
SnT, University of Luxembourg
Luxembourg, Luxembourg
jacques.klein@uni.lu

Yves Le Traon
SnT, University of Luxembourg
Luxembourg, Luxembourg
yves.letraon@uni.lu

ABSTRACT

Specific requirements of clients lead to the development of variants of the same software. These variants form a Software Product Line (SPL). Ideally, testing a SPL involves testing all the software products that can be configured through the combination of features. This, however, is intractable in practice since a) large SPLs can lead to millions of possible software variants and b) the testing process is usually limited by budget and time constraints. To overcome this problem, this paper introduces PLEDGE, an open source tool that selects and prioritizes the product configurations maximizing the feature interactions covered. The uniqueness of PLEDGE is that it bypasses the computation of the feature interactions, allowing to scale to large SPLs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Design, Experimentation

Keywords

Software Product Lines, Combinatorial Interaction Testing, Scalability, Search-based Approaches, Prioritization, T-wise

1. INTRODUCTION

Specific needs of particular clients lead to the development of several variants of the same software. These variants, which share a common set of features while having distinct functionalities form a Software Product Line (SPL). SPLs rely on variability modeling which uses Feature Models (FMs) as the standard and compact representation [8] of all the possible products of a SPL. By representing the combination of features and expressing constraints, FMs allow configuring tailored software products.

Testing a SPL can lead to millions of possible variants to test, (it can be shown that 270 independent and optional features can cover more products that there are atoms in the universe). It results in a situation which is inconceivable in an industrial context subjected to time and budget constraints. As a consequence, it is necessary to reduce that testing effort by selecting the relevant variants to test.

Combinatorial Interaction Testing [2, 10] (CIT) focuses on the interactions between t features (t-wise). It has been identified as a relevant approach to reduce the number of products for SPLs [4, 7, 13]. However, existing applications of CIT rely on the expensive computation of all the t-wise interactions, while considering the FM constraints. Although relying on efficient satisfiability (SAT) solvers, this problem is known to be NP-complete in the general case [7], limiting existing approaches to small features interactions, e.g., $t = 2$ or $t = 3$ for large SPLs.

This paper introduces PLEDGE, an open source Java tool that selects and prioritizes product configurations maximizing the number of t-wise interactions covered. The novelty of the tool is that it does not require to compute any feature interaction, thus allowing to scale to large SPLs and to any value of t [3]. It also allows specifying the desired number of product configurations to test.

The remainder of this paper is organized as follows. Section 2 present the research challenges underlying PLEDGE. Section 3 describes the approaches implemented by the tool and presents its features and architecture before giving a discussion. Finally, Section 4 concludes the paper.

2. RESEARCH CHALLENGES

The first research challenge is the *combinatorial explosion* of the number of products to consider. For instance, the FM of a video player [12] allows configuring more than 4.5×10^{13} different variants of this player. In that context, *how to test the SPL?* Testing all the possible products is intractable, leading to the necessity to reduce the number of products to test to a reasonable value while trying to maximize the level of confidence in the products that are tested.

The second challenge is the *scalability*. Existing techniques like [6, 7] are constraint-based, which makes the problem difficult to handle for large FMs, i.e., heavily-constrained FMs with more than 1,000 features. They also depend on

*FNRS Postdoctoral Researcher.

SPLC 2013 workshops, August 26 - 30 2013, Tokyo, Japan

the value of t and are thus limited to small feature interactions. The tool presented in this paper aims at providing solutions using heuristic instead of constraints and does not depend on the value of t, allowing to scale to large SPLs.

The last challenge concerns the *order in which the products should be tested*. Suppose we know the software products that have to be tested, e.g., the products that will be sold. If the testing budget allows testing only some of these products, *which products should be tested first? What are the most important products?*

3. THE PLEDGE TOOL

PLEDGE - a Product Line EDitor and tests GEneration tool is an open source[1] Java application of around 4,000 lines of code in its current version. It aims at solving the abovementioned challenges by both prioritizing and generating the product configurations to test.

3.1 Approach To Products Prioritization and Generation

PLEDGE implements approaches [3] that make use of a similarity heuristic (see Section 3.1.2) to both generate and prioritize product configurations derived from a FM of a SPL with respect to t-wise testing. The advantage of the heuristic is that it avoids computing any t-wise interaction.

3.1.1 T-wise Testing

T-wise testing focuses on the interactions between t features. At the FM level, we consider all the possible interactions between the features. Indeed, the FM describes the dependencies between all the features of the SPL, thus modeling all the possible variants that can be configured. It thus allows combining any t features together in a product variant.

With reference to the FM of Figure 1, a 2-wise combination of features is for instance $(Calls, GPS)$. The absence of a feature is also considered in a feature interaction. Thus, another example of such a 2-wise interaction might be $(Calls, \neg GPS)$. The t-wise interactions of a FM represents all the interactions between t features that are valid (i.e., which fulfill the constraints of the FM). For instance, the 3-wise interaction $(Calls, GPS, Basic)$ is invalid since the GPS feature excludes the $Basic$ one. The validity of a given combination can be checked with a SAT solver.

A product configuration, i.e., a test case is a configuration of the FM. It can be seen as a list of selected and unselected features. The selected features are supported by the software variant while the unselected features are not. To simplify, we will refer to product configurations as products.

Considering the t-combinations between selected and unselected features of a product, it is possible to know the interactions covered by this specific product. By knowing the t-wise interactions covered by products, i.e., the test suite, it is possible to evaluate the t-wise coverage of these products with respect to all the valid t-wise interactions of the FM. For instance, the following product P is a valid configuration of the FM and corresponds to a test that is selected or prioritized by the tool. Selected features are preceded with $+$ while unselected ones are preceded with the $-$ symbol.

$$P = \{+Mobile\ Phone, +Calls, -GPS, +Screen, +Basic, \\ -Colour, -High\ Resolution, -Media, -Camera, -MP3\}.$$

[1] `https://github.com/christopherhenard/pledge`.

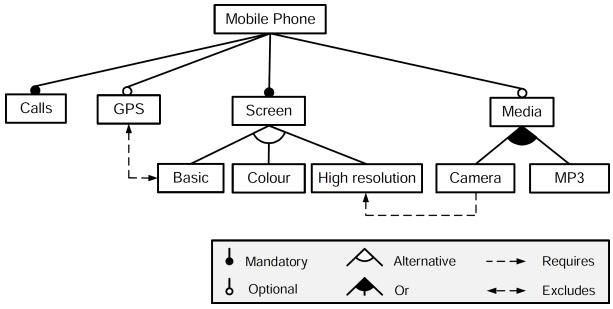


Figure 1: A simple feature model of a mobile phone product line [1], representing the features and their dependencies.

3.1.2 T-wise Products' Generation and Prioritization

The objective of the products' generation is to maximize the amount of t-wise combinations covered by the selected products. The prioritization aims at ordering products according to their ability to cover t-wise interactions.

Computing the t-wise combinations of features is computationally expensive. To overcome this problem, the prioritization and generation techniques [3] implemented by the tool bypass this problem. To this end, they consider products represented as a set of selected and unselected features. Based on this representation, a distance measure between two products has been introduced and serves as a similarity heuristic to compare the products. Experimental results [3] demonstrated the benefit of the similarity heuristic to mimic t-wise coverage.

To generate the products to test, the similarity heuristic was introduced as a fitness function to evaluate a set of products and to guide a search process. To this end, a SAT solver [11] was used to generate valid configurations of the FM forming the search space. Then, a search-based approach guided by the fitness function performed the selection of the products to test, replacing the worst product in terms of fitness by another product from the search space. The approach is executed during a user-specified amount of time and return the user-specified number of products required to test, which are prioritized on the fly. To prioritize a given set of products, two approaches called *Greedy* and *Near Optimal* were proposed [3]. They make use of the distance between the products to order them.

An experimental study conducted on 124 FMs from 11 features to $\approx$ 7,000 features demonstrated the benefit and scalability of these approaches [3]. For instance, Table 1 presents the t-wise coverage for $t = 2, ..., 6$ reached with 50 and 100 products generated for the Linux Kernel 2.6.28.6 FM (6,888 features) in 30 minutes on 10 independent executions. Regarding the prioritization, Figure 2 depicts the t-wise coverage difference between our approach and a random one on a set of 500 products, averaged on the 124 FMs studied on 10 independent executions. For instance, 14% of difference is observed with 100 products for $t = 6$.

In addition, the conducted experiments demonstrated that the test generation approach competes with existing t-wise tools, e.g., [6, 7] that are generally limited to small values of t (i.e., $t = 2$ or $t = 3$) while allowing to mimic t-wise for greater value of t. The proposed approaches are designed to handle large scale SPLs since they do not compute any

Table 1: T-wise coverage (%) achieved for the Linux Kernel 2.6.28.6 feature model (6,888 features) with 50 and 100 products generated in 30 minutes for 10 independent executions.

	50 products	100 products
2-wise	96.92%	97.71%
3-wise	91.96%	94.60%
4-wise	81.37%	88.53%
5-wise	64.42%	77.38%
6-wise	45.24%	61.13%

combination of features and do not depend on the value of t. Further experimental results are available in [3].

3.2 Functionalities and Usage

The current version of PLEDGE allows performing the following actions:

- Loading a FM from a file. PLEDGE supports the SPLOT [12] and DIMACS (Conjunctive Normal Form) formats,
- Visualizing the FM information, for instance, its constraints and features,
- Editing the FM, by adding or removing constraints,
- Generating the products to test from the FM, by specifying the number of products desired and the time allowed for generating them,
- Loading a list of products and prioritize them according to one of the two prioritization techniques proposed,
- Saving the generated or prioritized products to a file.

The tool can be used with both a command line and graphical user interface (GUI). Figure 3 show the GUI of PLEDGE. The command line interface eases the use of the tool in a scripting or automated context while the GUI is more user-friendly. In terms of usage, an HTML user guide is embedded in the application to help the user using the tool. In terms of development, the source code and the Javadoc documentation are available. Since the tool is open source, it can be used as a library as well.

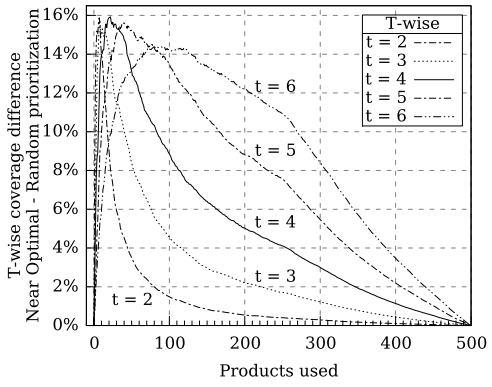


Figure 2: T-wise coverage difference with a random prioritization for 10 independent executions. The results are averaged on 124 feature models.

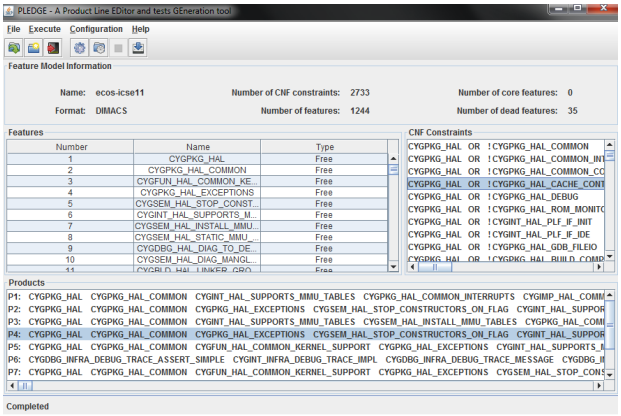


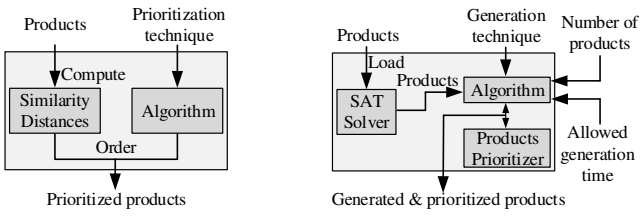
Figure 3: PLEDGE's graphical user interface.

3.3 Architecture

PLEDGE is built upon the Model-View-Controller (MVC) architecture [9]. This architecture allows separating the internal logic (the model) from the graphical representation (the view) and the user's interaction with it (the controller(s)). Concretely, the model is observed by the view, which graphically represents the model. When the user performs an action on the GUI, a controller acts on the model to change its internal representation which triggers the view to be refreshed. The main advantage of this architecture is the separation of concerns and the code reusability. The tool also makes use of common design patterns [14], like the *strategy* pattern to implement the products' prioritization and generation techniques, the *adapter* pattern to map the model of complex graphical component to the tool's model or the *observer* pattern to implement the MVC architecture.

The architecture of the products' prioritizer and generator units are presented in Figure 4. The products prioritizer (see Figure 4(a)) takes as input the list of products to prioritize and the prioritization technique to use. The prioritization technique is specified using the *strategy* design pattern. The prioritization unit is using the similarity distances computed on the products and the ordering technique to output the prioritized list of products.

The products' generator (see Figure 4(b)) takes as input the FM, the generation technique, the number of products to test and the time allowed for generating them. Internally, it uses a SAT solver to handle the FM constraints and to get valid products. The algorithm, specified by the generation technique via the *strategy* design pattern uses the allowed amount of time to generate the number of products specified.



(a) Prioritizer (b) Generator

Figure 4: PLEDGE's architecture.

3.4 Discussion

The products' generation technique implemented by the tool requires the number of products to generate and the amount of time allowed to generate them to be manually specified by the user. These parameters aims at making the testing process flexible. Indeed, other approaches, e.g., [7] generate all the products required to achieve 100% of coverage of the t-wise interactions. The inconvenience of these approaches is that they may take a large amount of time to perform this full coverage and they may generate too many products. To the authors' knowledge, the approach implemented by PLEDGE is a unique feature which aims at maximizing the t-wise coverage for the specified amount of products, using the specified amount of time. It thus gives a partial coverage but also makes the testing process more tractable with large SPLs. In addition, the prioritization of a given list of products is a unique feature of the tool.

Both the products' generation and prioritization methods scale to large FMs. While using constraints solver, we understood that solving constraints takes time and is an obstacle to scalability. We also found out that computing the t-wise coverage of products is not tractable for large FMs since all the t-wise combination of the products have to be considered. The products' generation approach makes use of a SAT solver only for generating valid products. Instead of using constraints, the proposed approaches are driven by a similarity heuristic. The benefit of this heuristic is that it mimics t-wise coverage and does not require to compute any combination of feature. It also does not depend on t. We show in [3], Section V.A the benefit of the heuristic in terms of computation.

Finally, while using a SAT solver to generate valid products, we found out that the generated products were predictable due to the fact that the constraints and the literal of the clauses are assigned values in a particular way. It means that one can guess the products that will be returned by the solver since it always assign values to the variable in same order. To overcome this problem, we randomized the way the solver assign values to its internal variables in order to get products from all the valid products' space.

4. CONCLUSION AND FUTURE WORK

We presented PLEDGE, a publicly available tool that allows generating and prioritizing product configurations of a SPL. It solves the challenge of testing a SPL in the context of Combinatorial Interaction Testing by selecting the products which maximize the t-wise interactions between features. By drastically reducing the number of products to test and ordering them while avoiding the combinatorial explosion induced by the computation of the t-wise interactions, it provides both a great level of flexibility and usability in a real and industrial testing process. This approach can also help generating products for model-based testing [5].

The current version of PLEDGE is the first release. The tool will be extended with additional features, including:

- The ability to draw a feature diagram and generating its boolean formula,
- The implementation of other prioritization and generation techniques via a plug-in system,
- The implementation of t-wise coverage computation and results visualizer,
- Integration as an Eclipse plug-in,
- Support of additional feature models format.

Finally, we invite researchers, developers and students to use our tool and/or to contribute to its development:

`http://research.henard.net/SPL/PLEDGE/`.

5. REFERENCES

[1] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, Sept. 2010.

[2] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG System: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[3] 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*, abs/1211.5451, 2012.

[4] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *ICSTW, A-MOST*, 2013.

[5] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Towards automated testing and fixing of re-engineered feature models. In *ICSE*, pages 1245–1248, 2013.

[6] A. Hervieu, B. Baudry, and A. Gotlieb. Pacogen: Automatic generation of pairwise test configurations from feature models. In *ISSRE*, pages 120–129, 2011.

[7] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of realistic feature models make combinatorial testing of product lines feasible. In *MODELS*, pages 638–652, 2011.

[8] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, Nov. 1990.

[9] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug. 1988.

[10] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, June 2004.

[11] D. Le Berre and A. Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, 7:59–64, 2010.

[12] M. Mendonca, M. Branco, and D. Cowan. S.p.l.o.t.: software product lines online tools, 2009.

[13] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Pairwise testing for software product lines: A comparison of two approaches. *Softw. Qual. Journal*, 2011.

[14] L. Rising. *The patterns handbook: Techniques, strategies, and applications*. 1998.