Towards Interactive Visualization Support for Pairwise Testing Software Product Lines

Roberto E. Lopez-Herrejon
Systems Engineering and Automation
Johannes Kepler University Linz, Austria
Email: roberto.lopez@jku.at

Alexander Egyed
Systems Engineering and Automation
Johannes Kepler University Linz, Austria
Email: alexander.egyed@jku.at

***Abstract*—Software Product Lines (SPLs) are families of related software products. SPL practices have proven substantial technological and economical benefits such as improved software reuse and reduced time to market. Software testing is a key development activity in SPLs, and it is uniquely challenging because of the usually large number of feature combinations present in typical SPLs. Pairwise testing is a combinatorial testing technique that aims at selecting products to test based on the pairs of feature combinations such products provide. Our previous work on evolutionary approaches for SPL testing and their comparative analysis has yielded a large amount of data that prompted us to explore ways by which to convey and represent this information. In this paper we present our early results in this effort. We describe three basic visualization applications to pairwise testing and highlight some of the open questions that we foresee. But most importantly, our driving goal is both to raise the awareness of the visualization problems in this area and to spark the interest of the software visualization community.**

I. Introduction

Software Product Lines (SPLs) are families of related software products, where each product provides a unique combination of *features* (i.e. increments in program functionality [1]). SPL practices have an extensive literature that attests to the substantial benefits they provide (e.g. [2]). A *feature model (FM)* represents all the possible feature combinations (typically a large a number) available in an SPL. The number of combinations poses a unique set of challenges because testing each individual product may not be technically or economically feasible.

Surveys and mapping studies on SPL testing [3], [4], attest to the increasing interest in testing within the SPL community. Among the SPL testing approaches are those based on *Combinatorial Interaction Testing (CIT)*. Their premise is to select a group of products where faults due to feature interactions are more likely to occur [5]. In CIT the focus has been mostly on *pairwise* interactions, meaning that these techniques consider the four possible combinations between any two features[1]. The combination of features in a product of an SPL determines the set of pairwise feature combinations that the product *covers*. Pairwise SPL testing aims to select a set of products such that their feature combinations cover the possible combinations of *all* interactions between two features according to the feature model of the SPL. This set is called a *covering array*.

[1]For `A` and `B` features: both selected, both not selected, `A` selected and `B` not, `A` not selected and `B` selected.

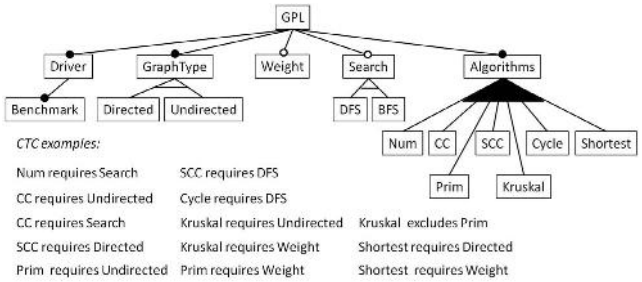

Fig. 1. Graph Product Line Feature Model

A recent survey of SPL use in industry showed that poor information visualization is an important and recurring problem [6]. Visualization techniques have been applied within the SPL context mostly for product configuration or in supporting program comprehension. The techniques used are not widespread and for the most part very fundamental. In this paper we describe three basic visualization applications to pairwise testing and highlight some of the open questions that we foresee. But most importantly, our driving goal is both to raise the awareness of the visualization problems in this area and to spark the interest of the software visualization community.

II. Feature Models and Running Example

A *feature model (FM)* represents all the possible feature combinations available in an SPL [7]. Feature models have become a *de facto* standard for modelling the common and variable features of an SPL and their relationships collectively forming a tree-like structure. The nodes of the tree are the features which are depicted as labelled boxes, and the edges represent the relationships among them. We use the *Graph Product Line (GPL)* [8], a standard SPL of basic graph algorithms, as our running example. Figure 1 shows the feature model of GPL.

A product in GPL has feature `GPL` (the root of the feature model) which contains its core functionality, a driver program (`Driver`) that sets up the graph examples (`Benchmark`) to which a combination of graph algorithms (`Algorithms`) are applied. The types of graphs (`GraphType`) can be either directed (`Directed`) or undirected (`Undirected`), and can optionally have weights (`Weight`). Two graph traversal algorithms (`Search`) are available: either Depth First Search (`DFS`) or Breadth First Search (`BFS`). A product must pro-

978-1-4799-1457-9/13/$31.00 ©2013 IEEE

vide at least one of the following algorithms: numbering of nodes in the traversal order (Num), connected components (CC), strongly connected components (SCC), cycle checking (Cycle), shortest path (Shortest), minimum spanning trees with Prim's algorithm (Prim) or Kruskal's algorithm (Kruskal).

In a feature model, each feature (except the root) has one parent feature and can have a set of child features. A child feature can only be included in a feature combination if its parent is included as well. The root feature is always included. In feature models there are four types of feature relations:

- *Mandatory features* are depicted with a filled circle and are selected whenever its respective parent feature is selected. For example, features Algorithms and Benchmark.
- *Optional features* are depicted with an empty circle and may or may not be selected if its respective parent feature is selected. An example is feature Search.
- *Exclusive-or relations* are depicted as empty arcs crossing over a set of lines connecting a parent feature with its child features. They indicate that exactly one of the child features must be selected whenever the parent feature is selected. For example, if feature GraphType is selected, then either feature Directed or feature Undirected must be selected.
- *Inclusive-or relations* are depicted as filled arcs crossing over a set of lines connecting a parent feature with its child features. They indicate that at least one of the features in the inclusive-or group must be selected if the parent is selected. If for instance, feature Algorithms is selected then at least one of the features Num, CC, SCC, Cycle, Shortest, Prim, and Kruskal must be selected.

Besides the parent-child relations, features can also relate across different branches of the feature model with the so called *Cross-Tree Constraints (CTC)*. Figure 1 shows the 13 CTCs of GPL's feature model (see [8]) in a textual manner. It should be noted though that there is not a standard visual representation of CTCs. For instance, Num requires Search means that whenever feature Num is selected, feature Search must also be selected. As another example Kruskal excludes Prim means that these two features cannot appear together in any product of the product line. These constraints, as well as those implied by the hierarchical relations between features, are usually expressed using propositional logic (in particular Conjunctive Normal Form clauses) and analysed with SAT solvers. For further details refer to [9].

III. Combinatorial Interaction Testing in SPLs

Combinatorial Interaction Testing (CIT) is a testing approach that constructs samples to drive the systematic testing of software system configurations [10]. When applied to SPL testing, the idea is to select a representative subset of products where interaction errors are more likely to occur rather than testing the complete product family [10]. In this section we provide the basic terminology of CIT for SPLs[2].

[2]Definitions based on [9] and [11].

Definition 1: Feature List (FL) is the list of features in a feature model.

Definition 2: A feature set, also called product in SPL, is a 2-tuple [sel,$\overline{sel}$] where sel and $\overline{sel}$ are respectively the set of selected and not-selected features of a member product. Let FL be a feature list, thus sel, $\overline{sel} \subseteq$ FL, sel $\cap$ $\overline{sel}$ = $\emptyset$, and sel $\cup$ $\overline{sel}$ = FL. The terms p.sel and p.$\overline{sel}$ respectively refer to the set of selected and not-selected features of product p.

Definition 3: A t-set ts is a 2-tuple [sel,$\overline{sel}$] representing a partially configured product, defining the selection of t features of the feature list FL, i.e. $ts.sel \cup ts.\overline{sel} \subseteq FL \wedge ts.sel \cap ts.\overline{sel} = \emptyset \wedge |ts.sel \cup ts.\overline{sel}| = t$. We say t-set ts is covered by feature set fs iff $ts.sel \subseteq fs.sel \wedge ts.\overline{sel} \subseteq fs.\overline{sel}$.

Definition 4: A t-set ts is valid in a feature model fm if there exists a valid feature set fs that covers ts.

Definition 5: A t-wise covering array tCA for a feature model fm is a set of valid feature sets that covers all valid t-sets denoted by fm. We also use the term test suite in this paper to refer to a covering array.

Most of the research on CIT for SPLs has focused on pairwise testing, i.e. t=2. Some of the techniques used are simulated annealing [12], evolutionary algorithms [13], constraint programming [14], and greedy ad hoc algorithms [11]. In the following section we illustrate these concepts and describe how our ongoing work on software visualization tries to address some of the challenges SPL pairwise testing entails.

IV. Visualizing SPL Pairwise Testing

Let us first explain the core concepts of pairwise testing. From the feature model in Figure 1, a valid 2-set is [{Driver},{Prim}]. It is valid because the selection of feature Driver and the non-selection of feature Prim do not violate any constraints. As another example, the 2-set [{GraphType,Search}, $\emptyset$] is valid because from the feature model it can be seen that both features can be selected without violating any constraint. Notice however that the 2-set [$\emptyset$,{Directed, Undirected}] is not valid. This is because feature GraphType is present in all the feature sets (mandatory children of the root) so always either Directed or Undirected must be selected. The covering array of GPL is then a list of products that covers *all* the possible 2-set, a total of 418. This list of products should be selected from the 73 possible valid products of GPL.

To help us experiment with the visualization tasks for SPL testing, we have developed a standalone tool prototype that receives data from our framework for covering array computation and comparison, and generates D3.js code for visualization on web pages [15]. We decided to use this tool as it is an easy entrance point for software visualization, and has a rich set of visualization techniques.

The first challenge was how to visualize covering arrays. For this purpose we have tried a tabular representation where each row represents a product of the covering array and each column represents each feature of the feature model. Features are colored, so an empty (blank) entry in the row of a product means that that feature is not selected in that product. Figure 2 shows an example of a covering array of GPL computed with

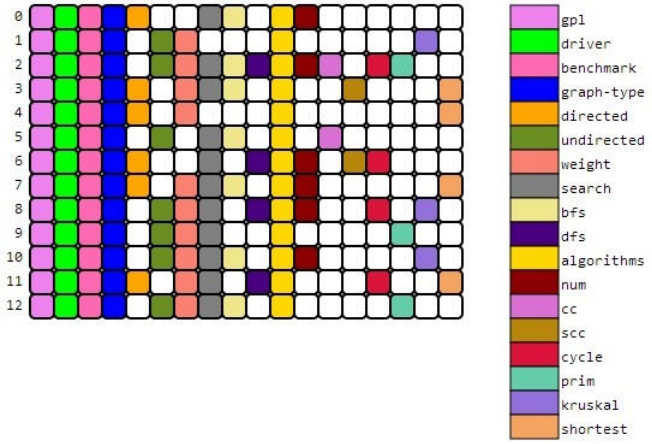

Fig. 2. Covering array example of GPL

a greedy approach [11][3]. In this figure it is easy to see that the first four features are always selected, as well as feature `Algorithms`. As another example, consider checking how the 2-set `[{GraphType,Search}` is covered. This can be done by simply inspecting the columns of both features. In our example almost all the products (except products 1 and 4) cover this 2-set.

The second challenge was to depict the actual 2-set coverage of the covering array. For this case we decided on using a combination of two techniques, bubble charts and heat maps [15]. We define one bubble for each of the 2-sets that need to be covered whose size and shade of color blue depended on the number of products of the covering array that covers each pair. Figure 3 shows our covering array example. The larger number was 13, meaning that all the products of the covering array covered the pair, for instance

[3]We depict the names of the features in lower case for readability.

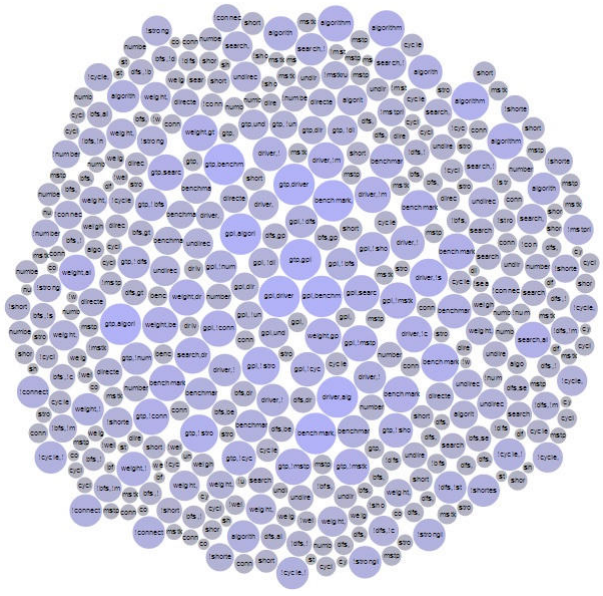

Fig. 3. Bubble and heatmap representation of covering array of GPL

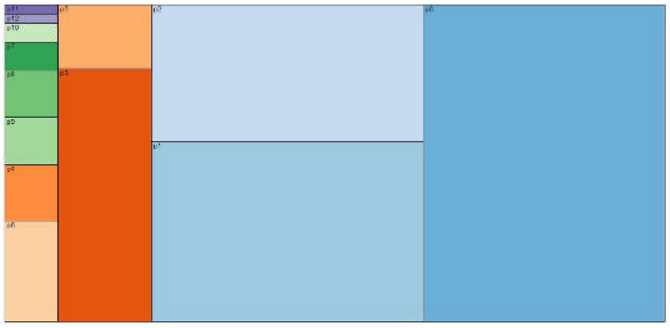

Fig. 4. Tree map representation of coverage increment for GPL

`[{GPL,Driver}, ∅]`, shown towards the center of the figure. By hovering over the bubbles, the users can see the 2-set names and the number of products they are covered by.

In the comparative analysis of testing approaches it is quite common to measure the increment in coverage that each new product of the covering arrays contributes. In other words, how many new 2-sets, over all the 2-sets, each product adds that have not been covered before. The third challenge we address was how to depict this metric. We experimented with percentage doughnuts but given that most of the times the increment percentages of the last products are usually small, their corresponding slices and labels were not intelligible. We experimented then with tree maps as shown in Figure 4.

V. Software Visualization and SPL

In this section we present an overview of the pieces of work most related to ours. Based on their focus, we can identify two main trends on the application of visualization techniques to SPLs: *i) product configuration* which is the process of creating a specific product of a product line, and *ii) program comprehension* which refers to the process engineers follow to perform, among others, software maintenance tasks.

In product configuration, one of the earliest works is by Nestor et al. [16]. They present a tool called VISIT-FC that provides basic visualization techniques such as color coding and incremental feature model browsing for supporting the users to configure large feature models as well as to understand design decisions such as feature costs. For this they depict features in different sizes or colors. Pleuss et al. present a more recent and small survey of the application of information visualization to SPL configuration [17]. They report that the most commonly used techniques have been clustering, decision trees, tree maps, cone trees, tables, and flow maps, and provide a short analysis of their advantages and drawbacks for product configuration. Furthermore, they outline some promising techniques, such as hyperbolic trees or space trees, and their research agenda. In a more recent work, Pleuss et al. [18] present the S2T2 Configurator. This tool displays feature models in a vertical layout and depicts CTCs as arcs between features depicted as nodes in a tree. Features can have different colors, sizes and icons to express the stage of the configuration process the user is at as well as to denote any inconsistencies detected or problems found. Nöhrer et al. have developed an interactive configuration tool called C02 that allows users to incrementally make decisions [19]. At each step, the sizes and the colors of both the already selected nodes

and those decisions that can still be taken are respectively adjusted to denote the decisions made and the likelihood that taking a decision will lead to a valid configuration in the shortest number of steps. In contrast with our work, all these research efforts are primarily focused on product configuration.

In program comprehension, the work by Apel et al. [20] presents a tool called FeatureVisu that uses clustering layouts to depict feature cohesion in the code base of SPLs. Their goal was to analyze this property in several SPL cases studies both implemented with preprocessor or advanced modularity approaches. The work by Feigenspan et al. [21] enhances pre-processor based software product lines with colors that denote which feature a line of code belongs to. They use a flexible and adaptable coloring scheme that exploits the fact that at any given moment only very few features appear together (or interact) in the code, so only a few colors are simultaneously needed which makes using colors an appealing option. Their field study showed that users performed better for some tasks when using colors and overall preferred this form of visualization. In contrast with our work, we do not focus on the source code and its mapping to feature models.

VI. Some Open Questions

We believe our early work just scratches the surface of the potential applications of visualization techniques for SPL testing. In this section we sketch some of the open questions our future work aims to address.

First and foremost is the issue of *scalability*. It is not uncommon to find industrial SPLs with hundreds or even thousands of features [22]. This fact poses the question: What is the right combination of pre-attentive properties most suitable for SPL testing? Changes in color or size might not be enough. A possibility is exploring other metrics (e.g. commonality [9]) to help the visualization.

Our ultimate goal is to provide users the capability to select, apply and analyze the results of SPL testing. To achieve that goal it is important to provide meaningful and interactive relationships between the different visualizations. This sparks the question: What are the appropriate interaction techniques for such tasks?

As with any other visualization work, it is important to assess the benefits and limitations of the techniques in real application scenarios. We plan to conduct experimental studies with both students and software professionals along the lines proposed by Wettel et al. [23].

Acknowledgements

This research is partially funded by the Austrian Science Fund (FWF) project P25289-N15 and Lise Meitner Fellowship M1421-N15.

References

[1] P. Zave, "Faq sheet on feature interaction," http://www.research.att.com/ pamela/faq.html.

[2] K. Pohl, G. Bockle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[3] E. Engström and P. Runeson, "Software product line testing - a systematic mapping study," *Information & Software Technology*, vol. 53, no. 1, pp. 2–13, 2011.

[4] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Information & Software Technology*, vol. 53, no. 5, pp. 407–423, 2011.

[5] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1883612.1883618

[6] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *VaMoS*, S. Gnesi, P. Collet, and K. Schmid, Eds. ACM, 2013, p. 7.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[8] R. E. Lopez-Herrejon and D. S. Batory, "A standard problem for evaluating product-line methodologies," in *GCSE*, ser. Lecture Notes in Computer Science, J. Bosch, Ed., vol. 2186. Springer, 2001, pp. 10–24.

[9] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.

[10] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Software Eng.*, vol. 34, no. 5, pp. 633–650, 2008.

[11] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *SPLC (1)*, E. S. de Almeida, C. Schwanninger, and D. Benavides, Eds. ACM, 2012, pp. 46–55.

[12] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.

[13] 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.

[14] A. Hervieu, B. Baudry, and A. Gotlieb, "Pacogen: Automatic generation of pairwise test configurations from feature models," in *ISSRE*, T. Dohi and B. Cukic, Eds. IEEE, 2011, pp. 120–129.

[15] M. Bostock, "D3.js website," 2013.

[16] D. Nestor, S. Thiel, G. Botterweck, C. Cawley, and P. Healy, "Applying visualisation techniques in software product lines," in *SOFTVIS*, R. Koschke, C. D. Hundhausen, and A. Telea, Eds. ACM, 2008, pp. 175–184.

[17] A. Pleuss, R. Rabiser, and G. Botterweck, "Visualization techniques for application in interactive product configuration," in *SPLC Workshops*, I. Schaefer, I. John, and K. Schmid, Eds. ACM, 2011, p. 22.

[18] A. Pleuss and G. Botterweck, "Visualization of variability and configuration options," *STTT*, vol. 14, no. 5, pp. 497–510, 2012.

[19] A. Nöhrer and A. Egyed, "C2o configurator: a tool for guided decision-making," *Autom. Softw. Eng.*, vol. 20, no. 2, pp. 265–296, 2013.

[20] S. Apel and D. Beyer, "Feature cohesion in software product lines: an exploratory study," in *ICSE*, R. N. Taylor, H. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 421–430.

[21] J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, M. Frisch, and G. Saake, "Supporting program comprehension in large preprocessor-based software product lines," *IET Software*, vol. 6, no. 6, pp. 488–501, 2012.

[22] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 73–82.

[23] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *ICSE*, R. N. Taylor, H. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 551–560.

[24] R. N. Taylor, H. Gall, and N. Medvidovic, Eds., *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 2011.