Searching the Variability Space to Fix Model Inconsistencies: A Preliminary Assessment

Roberto E. Lopez-Herrejon, Alexander Egyed

Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
roberto.lopez@jku.at, alexander.egyed@jku.at

Abstract. Recent years have witnessed a convergence between research in Software Product Lines (SPL) and Model-Driven Engineering (MDE) that leverages the complementary capabilities that both paradigms can offer. A crucial factor for the success of MDE is the availability of effective support for detecting and fixing inconsistencies among model elements. The importance of such support is attested by the extensive literature devoted to the topic. However, when MDE is coupled with SPL, the research focus has been devoted to inconsistency detection, while leaving fixing largely unexplored. To address this issue, a first step is locating where to apply the required fix(es) such that the necessary feature combinations of a product line are considered. It is not uncommon for the number of such feature combinations – variability space – to be quite large which renders unfeasible any exhaustive exploration. In this paper, we present early results of our ongoing work which relies on a basic search technique to effectively identify the places where the fixes should be placed. We evaluated our approach with sixty SPL examples.

1 Introduction

Today, software systems are more frequently being built as a family of systems also known as *Software Product Line (SPL)* [1,2]. In a SPL, each member product implements a different combination of *features* – increments in program functionality [1]. The success of a SPL lies at the effective management and realization of its *variability*, defined as the capacity of software artifacts to vary [3]. Extensive research and practice has been documented that corroborates the significant benefits of applying SPL practices both in academia and industry [2].

As Model-Driven Engineering practices become more pervasive, so does the importance of keeping all the involved models consistent. A core objective of research in *consistency checking* has been to verify that a model adheres to *consistency rules* that describe the semantic relationships amongst their elements. It should be pointed out that, in the context of this paper, a model is any software artifact such as source code, configuration scripts, UML models themselves, etc. A classical example of a consistency rule in UML is that if a sequence diagram has a message `m` targeting an object of class `C`, then the class diagram of class `C` must contain method `m`. Violations to consistency rules are called *inconsistencies* and must be effectively detected and, whenever

possible, resolved. Multiple approaches have been proposed for consistency checking that have proven successful.

Variability poses an even more stringent demand for consistency checking, namely, verifying that not only one but *all* possible feature combinations that are allowed in the product line – *variability space* – are indeed consistent. *Variability modeling* specifies all meaningful and legal feature combinations in a SPL, and its de facto standard are *Feature Models (FM)* [4]. A rather naive approach would thus be to check the consistency and fix any inconsistencies for each possible feature combination that is specified by a feature model. Not surprisingly, this trivial approach is unfeasible due to the large number (potentially exponential) of feature combinations. In this paper, we use variability modeling analysis ([5]) for guiding the search to effectively determine the places where consistency fixes should be placed. We evaluate our approach in sixty SPL examples.

2 Detecting Inconsistencies with Safe Composition

The main source of inconsistencies in models that have variability is the discrepancies between what variability is modeled (using a feature model) and how variability is actually realized. A mechanism to detect such discrepancies works by mapping to a propositional logic representation both the feature model and the consistency rule instances present in realization of a SPL. This representation is then used by SAT-based techniques for their analysis. In this section we briefly explain how this process works, for more details refer to [6].

Feature models. Feature models are the de facto standard to model the common and variable features of SPL and their relationships [4]. Figure 1 shows a feature model example. Features are depicted as labeled boxes and are connected with lines to other features with which they relate, collectively forming a tree-like structure. The root feature of a SPL is always included in all programs, in this case the root feature is `VOD`. A feature can be classified as: *mandatory* if it is part of a program whenever its parent feature is also part (e.g. `Play`), and *optional* if it may or may not be part of a program whenever its parent feature is part (e.g. `Record`). Mandatory features are denoted with filled circles while optional features are denoted with empty circles both at the child end of the feature relations denoted with lines. Features can be grouped into: *inclusive-or* relation whereby one or more features of the group can be selected and *exclusive-or* relation where exactly one feature can be selected. These relations are depicted as filled arcs and empty arcs respectively. In Figure 1, feature `Record` with its children features `CD` and `Card` is an example of inclusive-or, whereas feature `Play` with children `TV` and `Mobile` form an exclusive-or. Additionally, there are constraints

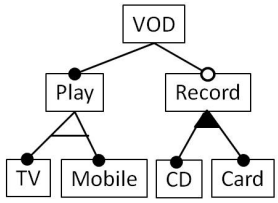

Fig. 1. Feature Model Example

that cannot be expressed directly on a feature diagram [5]. Each feature combination is called a *configuration*, defined as follows (adapted from [5]):

Definition 1. *A configuration conf 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 FS be the set of features of a feature model, such that sel, $\overline{sel} \subseteq$ FS, sel $\cap$ $\overline{sel}$ = $\emptyset$, and sel $\cup$ $\overline{sel}$ = FS. We use the terms conf.sel and conf.$\overline{sel}$ to respectively refer to the set of selected and not-selected features of conf, and $\emptyset_{conf}$ to denote the empty configuration [$\emptyset$,$\emptyset$].*

A configuration is *valid* if it adheres to the semantics of its feature model. For instance, the following is a valid configuration of our feature model in Figure 1: `conf=[{VOD, Play, TV}, {Mobile, Record, CD, Card}]`. It should be noted that typical feature models can have hundreds if not thousands of valid configurations.

Safe Composition. *Safe composition* comes from research in programming languages and is the guarantee that *all* valid configurations of a SPL's feature model are type safe, that is, they do not have undefined references to structural elements such as classes, methods or fields [7]. Safe composition is based on Czarnecki's et al. observation that variability realization should follow from variability modeling (i.e as denoted in feature models [8]). Safe composition uses propositional logic to express and relate these two terms. The mapping from feature models to propositional logic is based on the type of feature or relation. For example, optional feature `Record` is mapped to `Record` $\Rightarrow$ `VOD`, meaning that in a configuration if `Record` is selected then `VOD` should also be selected. As another example, inclusive-or relation is mapped to `Record` $\Leftrightarrow$ `CD` $\vee$ `Card`, which means that if `Record` is selected either or both of `CD` and `Card` can be selected. More details on this mapping can be found in [5]. Let $\texttt{PL}_f$ denote the propositional logic representation of a feature model. The second term is the variability realization which is computed for each consistency rule instance that we want to verify. We denote it with $\texttt{IMP}_f$. Our interest is in verifying that individual consistency rule instances are consistent for *all* valid configurations of the product line. This intent is captured by Equation (1). When this equation is evaluated with a *satisfiability, (SAT)* solver, if it is satisfiable it means that there are configurations that make the rule instance inconsistent. Such faulty configurations can be readily identified.

$$\neg(PL_f \Rightarrow IMP_f) \tag{1}$$

$$IMP_f \equiv F \Rightarrow \bigvee_{i=1..k} Freq_i \tag{2}$$

Let us explain how term $\texttt{IMP}_f$ can be obtained. Recall our UML consistency rule example: *Message action must be defined as an operation in receiver's class.* Assume that feature `Play` has in its sequence diagram message action `play` with class `Player` as receiver but that its class diagram does not define that method. For this instance to be consistent, the `play` method ought to be defined in other feature. Assume that it is defined in the class diagram of feature `TV`. Following Equation (2) our term is then $\texttt{IMP}_f \equiv$ `Play` $\Rightarrow$ `TV`. We define *requiring elements* as the set of elements that requires the presence of other elements to be consistent. For example, message `play` with its parameter and target lifeline. Also, we define *required elements* as the set of elements that make consistent a set of requiring elements. An example is method `play`.

3 Fixing Inconsistencies by Searching Variability Space

Let us consider our previous example with $\texttt{IMP}_f \equiv \texttt{Play} \Rightarrow \texttt{TV}$. When this term is passed to the safe composition Equation (1) and evaluated it is satisfiable. There are three configurations at fault, namely those that have feature `Mobile` selected. The question is: Where do fixes (definition of method play) should be placed? Before we proceed, we define some terms and functions our search algorithm depends on.

Definition 2. *A Consistency Rule Instance (CRI) is a 4-tuple [F,RME,TS,FC] where:*

- `F` is the feature that contains the requiring model elements, i.e. left-hand side term in $\texttt{IMP}_f$.
- `RME` are the requiring model elements.
- `TS` is the set of pairs (feature, `REDME`) which corresponds to the feature that contains the required model elements `REDME`. We refer to the set of features in the pairs of `TS` as `TS[feature]`.
- `FC` is a faulty feature configuration that violates the consistency rule instance.

For example, the CRI of message `play` is: `[Play, {`$\texttt{play}_{msg}$`}, {(TV,` $\texttt{play}_{op}$`)}, [{VOD, Play, TV},{Mobile, Record, CD, Card}]]`. Note that we use subscripts `msg` and `op` to respectively refer to the message use and operation definition. Now we define some other additional auxiliary functions:

- *pwc(P, F, G)*: pair-wise commonality operation that receives as input a feature model P and two features F and G, and returns the number of products that have both features.
- *maxCom(F,FC,TS,FS)*: returns a feature G such that G $\in$ FC.sel , G $\notin$ TS and G $\notin$ FS which has the highest pair-wise commonality value with F.
- *SafeComposition(*PL_f*,F,TS[feature])*: evaluates safe composition Equation (1) with $\texttt{IMP}_f \equiv F \Rightarrow \bigvee_{G \in TS[feature]} G$. Returns $\emptyset_{conf}$ if SAT evaluation is unsatisfiable, otherwise returns the first faulty configuration found.

Definition 3. *Fixing set: A fixing set for a CRI [F,RME,TS,FC] is a set of features FS such that SafeComposition(*PL_f*,F,FS)=*$\emptyset_{conf}$*. In other words, FS guarantees that CRI is consistent for all feature configurations.*

Algorithm (1) sketches our approach for computing minimal size fixing sets. It uses Breadth First Search with a queue to store the candidate partial fixing sets. For each fixing set, it selects candidate features that have the highest pair-wise commonality value with the corresponding requiring feature `F`. The intuition is that such candidate features are the most likely ones to appear in the same configurations where feature `F` appears and thus they have the higher chances to resolve the inconsistencies. Notice that our function `maxCom` chooses a feature from the unselected features of the current faulty configuration `FC` and that have not been already chosen in the fixing set `FS`. This process continues until no faulty configuration is found for one of the partial fixing sets stored in the queue. The loop terminates because in the worst case scenario all features in a SPL (except the requiring feature `F`) would be in the fixing set.

Algorithm 1 Computing Minimal Size Fixing Sets

Input: CRI of requiring type $[F, RME, TS, FC]$ with $FC \neq \emptyset_{conf}$, and PL_f.
Output: Fixing set FS.
$FC' := FC$
$FS := TS[feature]$
$FSQ.enqueue(FS)$
while $FC' \neq \emptyset_{conf}$ **do**
  $FSQ.dequeue(FS)$
  $G := maxCom(F, FC', TS, FS)$
  $FS := FS \cup G$
  $FC' := SafeComposition(Pl_f, F, FS)$
  $FSQ.enqueue(FS)$
end while
return FS

4 Preliminary Evaluation

To evaluate our algorithm we gathered 60 feature models publicly available in the SPLOT repository (a website that collects features models and is open to the community [9]), and computed fixing sets for all the features in all the features models by calling our algorithm with an empty parameter `TS` and a `maxCom` scheme of not considering SPL-wide common features. The sizes of the feature models analyzed range from 9 to 94, and number of configurations from one to millions.

The first step in our evaluation consisted in measuring the time it took to compute the pwc values. For this purpose we use FAMA tool because it permits definition of feature model operations [10]. Additionally, it provides support for different reasoning engines, in our case we utilized Binary Decision Diagrams (BDD) which are more appropriate for the implementation of counting operations such as ours. We ran our examples on an Intel Core-Duo at 2.8 GHz. As expected, the computation time increases steadily as the number of features increases (with 94 features it took 1600 secs aprox.). It should be noted though that this computation is performed only once and could be carried out in a lazy form as needed thus mitigating this computation time. Part of our future work is exploring more efficient approaches to compute these values [11]. Furthermore we observed that, sauf of few exceptions, even for larger feature models the length of the majority of the fixing sets is mostly around five elements and the corresponding computation time is in the order of miliseconds even for the largest feature models.

5 Conclusions and Future Work

In this paper, we presented a simple search algorithm that finds the features where the required elements of single consistency rule instances must be located to guarantee consistency for all the valid feature configurations of a product line. We analyzed our algorithm with sixty feature model examples and found that it performs efficiently and can scale.

We argue that Software Product Lines is a research area ripe for Search-Based Software Engineering. This is so because of the sheer number of possible feature configurations that typical feature models represent (which renders unfeasible considering the entire search space), and the fact that the desired solutions are usually not unique or optimal so a form of fitness function must be employed [12]. A first work that draws this connection is Ullah's which combines a genetic algorithm with a clustering technique for evolving a single system into a SPL aiming at causing the least impact on the underlying system architecture [13].

Currently we are working on fixing multiple consistency rule instances. The main challenge is when instances overlap such that fixing one inconsistency may cause new inconsistencies in another rule instance in the same or in difference feature configurations. We are exploring basic Hill-Climbing search and other different metaheuristics for addressing this problem.

Acknowledgments. We thank Alexander Nöhrer, Pablo Trinidad and David Benavides for their tool support. We also thank our anonymous reviewers for their references. This research was partially funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965.

References

1. Batory, D.S., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Software Eng. **30**(6) (2004) 355–371
2. Pohl, K., Bockle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)
3. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw., Pract. Exper. **35**(8) (2005) 705–754
4. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
5. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. **35**(6) (2010) 615–636
6. Lopez-Herrejon, R.E., Egyed, A.: Detecting inconsistencies in multi-view models with variability. In Kühne, T., Selic, B., Gervais, M.P., Terrier, F., eds.: ECMFA. Volume 6138 of Lecture Notes in Computer Science., Springer (2010) 217–232
7. Thaker, S., Batory, D.S., Kitchin, D., Cook, W.R.: Safe composition of product lines. In Consel, C., Lawall, J.L., eds.: GPCE, ACM (2007) 95–104
8. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L., eds.: GPCE, ACM (2006) 211–220
9. : Software Product Line Online Tools(SPLOT) (2011) http://www.splot-research.org/.
10. : FAMA Tool Suite (2011) http://www.isa.us.es/fama/.
11. Fernández-Amorós, D., Gil, R.H., Somolinos, J.A.C.: Inferring information from feature diagrams to product line economic models. In Muthig, D., McGregor, J.D., eds.: SPLC. Volume 446 of ACM International Conference Proceeding Series., ACM (2009) 41–50
12. Harman, M.: Why the virtual nature of software makes it ideal for search based optimization. In Rosenblum, D.S., Taentzer, G., eds.: FASE. Volume 6013 of Lecture Notes in Computer Science., Springer (2010) 1–12
13. Ullah, M.I.: Cope+: A method for design and evaluation of product variants. Technical Report SERG-2009-03 (August 2009)