Towards Fixing Inconsistencies in Models With Variability

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

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

ABSTRACT

Recent years have witnessed a convergence between research in 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 coupled with variability, the research focus has been devoted to inconsistency detection, while leaving the important issue of fixing the inconsistency largely unaddressed. In this research-in-progress paper, we explore one of the issues that variability raises for inconsistency fixing. Namely, in which features to locate the fixes. We compute what is the minimal number of fixes and use it as a baseline to compare fixes obtained with a heuristic based on feature model analysis and random approaches. Our work highlights the pros and cons of both approaches and suggests how they could be addressed.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Model, Consistency, Software Product Line

Keywords

Consistency checking, variability, safe composition, Feature Oriented Software Development

1. INTRODUCTION

As Model-Driven Engineering practices become more commonplace, so does the importance of keeping all the involved models consistent. A core objective of research in *consistency checking* has been the verification that models adhere to *consistency rules* which describe the semantic relationships amongst their elements [23]. 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 [23].

VaMoS'12 January 25-27, 2012 Leipzig, Germany

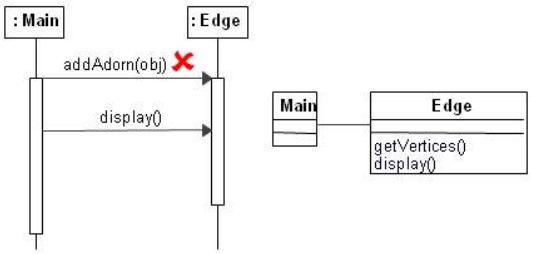

Figure 1: Standard Model Inconsistency Example

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 are indeed consistent. *Variability modeling* specifies all meaningful and legal feature combinations in a SPL, and its de facto standard are *Feature Models (FM)* [12]. 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. By fixing, we mean changing the model (e.g. adding, removing or modifying model elements) such that consistency rules are no longer violated. Not surprisingly, this trivial approach is unfeasible due to the large number of feature combinations.

In this research-in-progress paper, we explore one of the issues that variability raises for inconsistenct fixing. Namely, the question we address is: *Where should the fixes be placed?* We compute what is the minimal number of fixes required for a consistency rule instance, and use it as a baseline to compare two fixing approaches: a heuristic approach based on feature model analysis and a random approach. Our work highlights the pros and cons of both approaches and suggests how they could be addressed.

2. RUNNING EXAMPLE

There has been extensive research on fixing model inconsistencies in recent years [15, 10, 9, 30]. For sake of simplicity, our running example uses basic graph algorithms knowledge in conjunction with standard UML models and constraints. A typical UML consistency rule is like the following: *A message action must be defined as an operation in the receiver's class.* For example, consider Figure 1. It shows two instances of this consistency rule: one for message `addAdorn` and one for message `display`. The

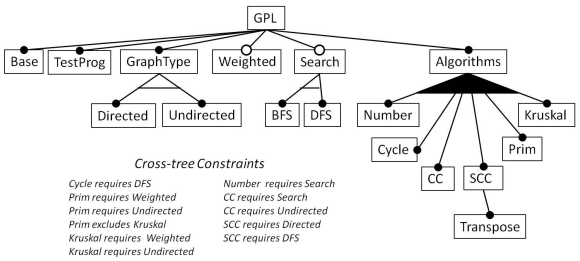

Figure 2: Graph Product Line Feature Model

first instance is inconsistent (marked with ✘) because the method `addAdorn` is not defined in the receiver's class. The second instance is consistent because method `display` is defined in class `Edge` of the class diagram.

DEFINITION 1. *In the context of a consistency rule instance, we identify two sets of elements:*

- *Requiring model elements: is a set of model elements that requires the presence of other model elements to be consistent.*
- *Required model elements: is a set of model elements that make a set of requiring model elements consistent.*

In our previous example, message `addAdorn` with its parameter and target lifeline are the requiring elements while method `addAdorn` and its parameter in class `Edge` are the required elements. This distinction is based on the fact that the constraint rule is defined in OCL from a message context.

As our running example we use a version of the *Graph Product Line (GPL)* [21], a standard problem for the analysis of SPL methodologies. The features in GPL are basic graph algorithms and data structures. A program in GPL is a combination of different algorithms implemented on different data structures. Features `Base` and `TestProg` provide the basic housekeeping functionality of this product line. Graphs (`GraphType`) can be of two types: `Directed` or `Undirected`. Optionally, the graphs can have weight (`Weighted`). Some algorithms require searches (`Search`) that can be either Depth-First Search (`DFS`) or Breadth-First Search (`BFS`). The algorithms (`Algorithms`) supported are: vertex numbering (`Number`), cycle checking (`Cycle`), Connected Components (`CC`) and Strongly Connect Components (`SCC`), and minimum spanning trees (`Kruskal` and `Prim`). The details of the algorithms are not relevant for our illustration purposes; for further information, please consult [21].

Feature models are the de facto standard to model the common and variable features of SPL and their relationships [12]. Figure 2 shows the feature model of our running example. Features are depicted as labeled boxes and are connected with lines to other features with which they relate, collectively forming a tree structure. The root feature of a SPL is always included in all programs, in this case the root feature is `GPL`. A feature can be classified as: *mandatory* if it is part of a program whenever its parent feature is also part (e.g. `Base` or `TestProg`), and *optional* if it may or may not be part of a program whenever its parent feature is part (e.g. `Weighted` or `Search`). Mandatory features are denoted with filled circles and 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 (e.g. `Algorithms` with its six children), and *exclusive-or* relation where exactly one feature can be selected (e.g. `Directed` or `Undirected`). These relations are depicted as filled arcs and empty arcs, respectively. Additionally, there are constraints that cannot be expressed directly on a feature diagram. These constraints are defined separately and are called *cross-tree constraints* [8]. As an example, the `Prim` excludes the `Kruskal` algorithm, which means that only one can be selected in a program. As another example, feature `Cycle` requires feature `DFS`. This means that if the `Cycle` is selected in a program, feature `DFS` must also be selected.

Each program of the SPL is called a *configuration*, which we defined as follows (adapted from [8]):

DEFINITION 2. *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 *valid configuration* meets all the restrictions imposed by the feature model including the cross-tree constraints. For example, the GPL program that provides only the `Kruskal` algorithm has the following valid configuration:

$$\begin{aligned} conf = \; & [\{GPL, TestProg, GraphType, Algorithms, \\ & Kruskal, Undirected, Base, Weighted\}, \\ & \{Directed, Search, DFS, BFS, Number, \\ & Cycle, CC, SCC, Prim, Transpose\}] \end{aligned} \tag{1}$$

Let us now add variability to our models of Figure 1 with relation to our SPL example. Assume that: *i)* the sequence diagram and method `display` are part of the `Weighted` feature, *ii)* the `Main` and `Edge` classes with their association and method `getVertices` are part of feature `Base`, *iii)* method `addAdorn` is part of feature `Undirected`.

We illustrate our work using *Feature Oriented Software Development (FOSD)*, a SPL compositional approach that provides formalisms, methods, languages and tools for building variable, customizable and extensible software. Nonetheless, our work can be applied to other compositional or integrative approaches[1]. Figure 3 shows our consistency example of Figure 1 in FOSD. Each feature is modularized in the so-called *feature modules* that contain all the artifacts that realize the corresponding feature. For further details on FOSD, please see [4, 7, 6].

The main challenge of fixing inconsistencies in models with variability is guaranteeing that inconsistencies are fixed for *all* the valid configurations of a SPL. The naive approach of creating all the valid configurations and individually checking their consistency is not feasible as the number of configurations can be extremely large. Even for our small running example, we would need to check its 62 products.

The crucial point to achieve this guarantee is to effectively determine where the required elements of a consistency instance must be. In other words and in FOSD terms, the feature modules where the required elements should be present so that they make every valid configuration consistent.

In the next section, we briefly summarize previous work on detecting inconsistencies in models with variability [22]. We built upon this work to analyze and compare two approaches to find the feature modules where fixes should be placed.

[1]Also known as annotative or negative variability [17].

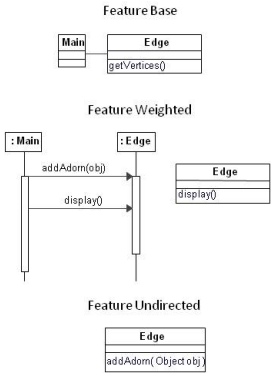

Figure 3: Feature Oriented Software Development Example

3. DETECTING INCONSISTENCIES

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 (via an integrative or compositional approach). 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 the realization of an SPL. This representation is then used by SAT-solvers for their analysis. Next we briefly explain how this process works for compositional approaches [22, 28], though integrative approaches follow a similar process [13].

3.1 Safe Composition

In the realm of compositional approaches, *safe composition* is the guarantee that *all* programs, which can be composed according to a SPL's feature model, are type safe, i.e. they do not have undefined references to structural elements such as classes, methods or fields [28]. Though most of the research has been done within the context of programming languages [14], the principles underlying safe composition can also be applied to other software artifact types as highlighted by our previous work [22].

Safe composition is based on Czarnecki's et al. observation that variability realization should follow from variability modeling [13]. Let $\texttt{DOM}_f$ denote all possible configurations that can be expressed in a feature model, and $\texttt{IMP}_f$ denote a variability realization consistency rule instance. Safe composition uses propositional logic to express and relate these two terms. Our interest is in verifying that *all* members of the product line satisfy individual consistency rule instances or, more concisely, that there is no member of the SPL that does not satisfy them. This is expressed as follows:

$$\neg\,(\texttt{DOM}_f \Rightarrow \texttt{IMP}_f) \tag{2}$$

For each consistency rule instance that we want to verify in a SPL realization, an expression of $\texttt{IMP}_f$ is computed and Equation (2) is evaluated with a *satisfiability (SAT)* solver [18]. When this equation is satisfiable, it means that there is at least one product line members that does not meet the consistency rule instance denoted by $\texttt{IMP}_f$. The inconsistent members can be readily identified by inspecting the values obtained by the SAT solver. We show next how the propositional formulas for $\texttt{DOM}_f$ and $\texttt{IMP}_f$ can be obtained.

3.2 Obtaining $\texttt{DOM}_f$

There exists extensive research on mapping feature models to propositional logic [5, 8]. This mapping is summarized in Figure 4a, where `P` and `Ci` are parent and child features, and `F1` and `F2` are

$$\begin{aligned}
mandatory: &\quad P \Leftrightarrow C \\
optional: &\quad C \Rightarrow P \\
exclusive-or: &\quad (C1 \Leftrightarrow (\neg C2 \wedge \neg C3 \wedge P)) \wedge \\
&\quad (C2 \Leftrightarrow (\neg C1 \wedge \neg C3 \wedge P)) \wedge \\
&\quad (C3 \Leftrightarrow (\neg C1 \wedge \neg C2 \wedge P)) \\
inclusive-or: &\quad P \Leftrightarrow C1 \vee C2 \vee C3 \\
requires: &\quad F1 \Rightarrow F2 \\
excludes: &\quad F1 \Rightarrow \neg F2
\end{aligned}$$

(a) *Propositional logic mapping.*

$$\begin{aligned}
root: &\quad GPL \Leftrightarrow true \\
mandatory: &\quad GPL \Leftrightarrow Base \\
optional: &\quad Weighted \Rightarrow GPL \\
exclusive-or: &\quad (BFS \Leftrightarrow \neg DFS \wedge Search) \wedge \\
&\quad (DFS \Leftrightarrow \neg BFS \wedge Search) \\
inclusive-or: &\quad Algorithms \Leftrightarrow Number \vee Cycle \vee \\
&\quad CC \vee SCC \vee Prim \vee Kruskal \\
requires: &\quad Cycle \Rightarrow DFS \\
excludes: &\quad Prim \Rightarrow \neg Kruskal
\end{aligned}$$

(b) *GPL examples.*

Figure 4: Feature models and propositional logic.

any features of cross-tree constraints. Notice that for brevity, the figure shows exclusive and inclusive ors mappings with only three children, but they can have any number of children as illustrated shortly. The term $\texttt{DOM}_f$ is the conjunction of all the propositional expressions computed as illustrated for our running example GPL in Figure 4b. The first example shows the special case of the root feature, it is defined this way because the root is part of all configurations. The second example illustrates mandatory feature `Base`, while the third one illustrates the optional feature `Weighted`. To illustrate the exclusive-or relation, we use the one that contains features `Search`, `BFS` and `DFS`. The only inclusive-or relation in our example is the one formed with `Algorithms` and the six graph algorithms we consider. Lastly, we illustrate both requires and excludes cross-tree constraints.

3.3 Computing $\texttt{IMP}_f$

Consistency rules describe the semantic relationships that must hold amongst the different elements of the views. Consistency rules are usually specified as well-formedness rules [26], or emerge as standard best practices in certain domains [16]. In this paper, our focus is on *requiring rules* [22], which are rules that assert the presence of structural elements that a feature requires.

The fixing approaches that we analyze work under certain assumptions. As we proceed, we shall explain them. The first one is the following:

ASSUMPTION 1. *Single feature containment: The requiring elements and the required elements are contained in single features.*

Recall our rule presented in Section 2: *Message action must be defined as an operation in receiver's class.* As we discussed before, in Figure 3, we see that there are two instances of this rule: *i)* for method `display`, and *ii)* for method `addAdorn`. The first instance is already consistent within feature `Weighted`, that is, method `display` is already defined in class `Edge`. In other words, both the requiring and the required elements are within a feature, which just happens to be the same. For the second instance, we have that the requiring elements (message `addAddorn`) are all contained in feature `Weighted` and the required elements (method `addAdorn`) are all contained in feature `Undirected`.

Next, we present how the propositional logic representation of a constraint instance is obtained.

Propositional logic representation. Let F be a feature that contains the requiring model elements. For a system program that includes feature F, it must therefore also include at least one other feature $\texttt{Freq}_i$ where the required elements are defined. This is denoted in the following expression[2]:

$$IMP_f \equiv F \Rightarrow \bigvee_{i=1..k} Freq_i \quad (3)$$

When feature F requires elements not defined in any other features, that is expression $\bigvee \texttt{Freq}_i$ evaluates to false, it means that such an element is not defined in the entire product line. This situation is clearly an error and renders it unnecessary to verify this constraint with the SAT solver.

By substituting $\texttt{IMP}_f$ in Equation (2), we obtain the logical expression that is passed to the SAT solver. In this case, it is the conjunction of all the terms of the features that define an element that feature F requires.

$$\neg(PL_f \Rightarrow IMP_f) \equiv PL_f \wedge F \bigwedge_{i=1..k} \neg Freq_i \quad (4)$$

This equation requires that the feature model to be non-void (i.e. to have at least one feature configuration), otherwise there would be no products to fix and the equation would never be satisfiable. This follows because the term $\texttt{PL}_f$ would be false if the feature model is void.

Let us now revisit our constraint instance example for message addAddorn. We have that $\texttt{IMP}_f \equiv$ Weighted⇒Undirected, because the requiring elements are contained in feature Weighted while the required elements are in feature Undirected. When our instance example is evaluated according to Equation (4), the SAT solver finds the equation satisfiable. This means that there is at least one feature configuration that makes the instance inconsistent. The features that were selected and those that were not selected can be determined by looking at the boolean assignment that the SAT solver made. In our example, one such configuration has feature Directed selected. Intuitively, this can be understood by considering that GPL graphs can either be directed or undirected. Thus, what this result indicates is that any configuration where graphs are directed would render our constraint instance inconsistent because it would not contain the required model elements. In the next section, we elaborate how an inconsistency like this can be fixed.

4. FIXING INCONSISTENCIES

As stated before, our work analyzes two approaches for finding feature(s) where the required elements of a constraint instance should be placed to guarantee that for *all* valid feature configurations the instance is consistent. We make the following assumptions.

ASSUMPTION 2. *Fixing Individual Inconsistency Instances: The objective is finding for individual inconsistency rule instances the features where to place their required elements. Thus, the side effects that fixing an inconsistency can have in other instances are not considered.*

[2]For notational simplicity in the rest of the paper, we overload feature terms such as F or $\texttt{Freq}_i$ to mean propositional logic terms and the set of software artifacts within a feature. We make the distinctions explicit when necessary.

Table 1: Pair-wise Commonality values of Weighted

Feature	PWC	Feature	PWC	Feature	PWC
GPL	44	Undirected	36	Kruskal	13
Base	44	DFS	30	BFS	12
Algorithms	44	Number	23	Directed	8
GraphType	44	CC	18	SCC	4
TestProg	44	Cycle	16	Transpose	4
Search	42	Prim	13		

ASSUMPTION 3. *Composition idempotence: The underlying SPL compositional approach is idempotent.*

In our context, this assumption implies that if two or more occurrences of the required model elements are composed only one remains in the composition result.

ASSUMPTION 4. *Composition monotonicity: The underlying SPL compositional approach is monotonic.*

In other words, the model elements in the features can only be added but not removed.

4.1 Preliminary Definitions

In this subsection we present more precise definitions of the terms and operations upon which we describe our fixing strategies (adapted from [8]).

DEFINITION 3. *Operation filter receives as input a feature model with a feature set FS and a configuration conf that can be partial, conf.sel ∪ conf.$\overline{sel}$ ⊆ FS. Filter returns the set of member products whose configurations prod satisfy conf.sel ⊆ prod.sel and conf.$\overline{sel}$ ⊆ prod.$\overline{sel}$.*

Consider operation filter applied to our GPL example with configuration [{Undirected},∅]. The result is the set of products that have feature Undirected selected, a total of 46.

DEFINITION 4. *Pair-wise commonality operation, pwc, receives as input a feature model P and two features F and G, and returns the number of products that have both features. It is defined as pwc(P, F, G) = |filter(P, [{F, G},∅])|.*

As example, the pwc values of feature Weighted are shown in Table 1. There are several insights that are revealed by these pwc values:

- Weighted appears in 44 products. This is because the root feature GPL is always included when Weighted is included and the pwc value of GPL is 44.
- Features Base, Algorithms, GraphType, and TestProgram are SPL-wide common features; that is, they appear in all the product line members. This can be inferred from pwc values because these features have the same values as the root GPL.
- Of the 44 products with feature Weighted, 36 use Undirected feature while 8 use the Directed feature. Notice that these latter products are precisely those found to be inconsistent in our example at the end of last section.

DEFINITION 5. *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 *target set* which is a set of pairs (feature, REDME) and represents 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 constraint rule instance of message addAdorn is: [Weighted, {$\texttt{addAdorn}_{msg}$}, {(Undirected, $\texttt{addAdorn}_{op}$)},[{Directed, Base, GPL, Algorithms, ...},{Undirected, ...}]].

Note that we use subscripts msg and op to respectively refer to the message use and operation definition. Also, for brevity we elided features of the faulty feature configuration.

DEFINITION 6. *Function SafeComposition(DOM_f,F, FS) receives as input the propositional logic representation of a feature model DOM_f, a feature F, and a list of features FS and evaluates Equation (4) with $\texttt{IMP}_f \equiv F \Rightarrow \bigvee_{G \in FS} G$. Returns $\emptyset_{conf}$ if SAT evaluation is unsatisfiable, otherwise returns the first faulty configuration found.*

DEFINITION 7. *Fixing set: A fixing set for a CRI [F,RME,TS,FC] is a set of features FS such that SafeComposition($\texttt{DOM}_f$,F,FS)=$\emptyset_{conf}$. In other words, FS guarantees that CRI is consistent for all feature configurations.*

4.2 Fixing Approaches

Our generic algorithm for computing fixing sets is shown in Algorithm (1). This algorithm computes a new set of target features FS where the required elements would be placed. It does so by iteratively choosing candidate features according to a fixing approach fixApp. Function fixApp must choose a feature from the unselected features of the current faulty configuration FC and that have not been already chosen in the fixing set FS. The loop iterates until no faulty configuration is found. 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 Generic Fixing Set Computation

Input: CRI of requiring type $[F, RME, TS, FC]$ with $FC \neq \emptyset_{conf}$, DOM_f, and a fixing approach $fixApp$.
Output: Fixing set FS.
$FC' := FC$
$FS := TS[feature]$
while $FC' \neq \emptyset_{conf}$ **do**
  $G := fixApp(F, FC', TS, FS)$
  $FS := FS \cup G$
  $FC' := SafeComposition(DOM_f, F, FS)$
end while

Let F be a feature, FC a faulty configuration and TS the target set of a CRI, and FS a set of features. We now define our two fixing approaches.

FIXING APPROACH 1. *Maximum Pairwise Commonality, maxComm(F,FC,TS,FS), computes a feature G such that G ∈ FC.sel , G ∉ TS[feature] and G ∉ FS which has the highest pair-wise commonality value with F.*

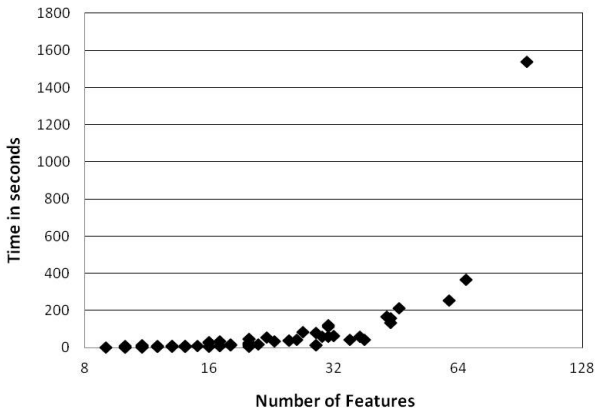

Figure 5: Pair-wise commonality computation time

The intuition behind this heuristic approach is that the candidate features with highest pairwise commonality are the most likely ones to appear in the same configurations where feature F appears and thus they have the highest chances to resolve the inconsistencies when the required elements are placed in them.

Recall that our CRI for method addAdorn was inconsistent because our current TS[feature] set only contained feature Undirected, leaving out the configurations where Directed is selected. Consider now the pwc values shown in Table 1. According to this table, the first candidate feature for the fixing set is feature GPL, being the root and therefore having the highest pwc value. This illustrates a potential caveat of this fixing approach. Certainly, placing the required model elements in the root feature will definitively resolve any inconsistencies, but it could be perhaps not the solution a SPL designer would desire.

The implementation of this approach could be fine tuned to follow different selection schemes, for instance not selecting the root feature. Another scheme could be not selecting SPL-wide common features. This latter scheme helps to illustrate a second caveat of our approach. Based on Table 1 the candidate fixing feature is Search with a pwc value of 42. Though adding this feature to the fixing set effectively resolves all the inconsistencies, it creates an overlap in six products that have both feature Search and Undirected. From the domain knowledge of GPL (i.e. graphs are either directed or undirected) we could see that a better solution is feature Directed.

FIXING APPROACH 2. *Random Compatible Feature, ranComp(F,FC,TS,FS), ranComp randomly selects a feature G such that G ∈ FC.sel, G ∉ TS[feature] and G ∉ FS and F appears at least in one configuration with G (compatible).*

The rationale of this approach is gauging if it pays off to compute the pairwise commonality information when finding fixing sets. Thus this approach serves as a counterbalance to this first heuristic one.

5. EVALUATION AND ANALYSIS

To evaluate our algorithm we proceeded as follows. We gathered 45 feature models that are available in the SPLOT repository [2], and computed with both approaches the fixing sets for all the features in all the features models using an empty parameter TS. In the

Algorithm 2 Computing Minimal Size Fixing Sets bfsMin

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**
  $FSQ.dequeue(FS)$
  $G := maxCom(F, FC', TS, FS)$
  $FS := FS \cup G$
  $FC' := SafeComposition(DOM_f, F, FS)$
  $FSQ.enqueue(FS)$
end while
return FS

case of `maxComm` we implemented the scheme of not considering SPL-wide common features.

First, we measured the time to compute the pwc values. For this purpose we use FAMA tool because it permits definition of feature model operations on different reasoning engines [1]. We utilized Binary Decision Diagrams (BDD) [18], 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. The results are depicted in Figure 5. Clearly, pair-wise commonality computation time increases steadily as the number of features increases. It should be noted though that this computation is performed only once and could be carried out in a lazy form as needed and memoized to implement `maxComm` more efficiently.

To compare and contrast both fixing approaches we computed the fixing sets of minimal length following Algorithm (2). This algorithm uses a Breadth-First search strategy storing the partial (incomplete) fixing sets in a queue. Notice that because we are interested only in the size of the fixing sets, it is indistinct which fixing approach is used, either `maxComm` or `randComp`.

Figure 6a shows the average lengths of the fixing sets using approaches `maxComm` and `randComp`, with Algorithm (2) as a baseline, denoted `bfsMin` in the figure. The values of `bfsMin` are sorted in increasing order across the x-axis for the 45 SPLOT models analyzed, and the values of `randComp` are the average of ten runs. The first thing that can be noticed is that for the majority of the SPLs, the average length of the fixing sets fluctuates around five places. This trend seems to suggest that the number of required fixing places does not solely relate to the number of features or of products. Also, as expected, the pairwise commonality approach `maxComm` selected on average shorter fixing sets than the random approach `randComp`, which indicates that the proposed heuristic does indeed work.

Figure 6b presents the execution time of approaches `maxComm` and `randComp` as a percentage of their counterpart execution time of the baseline `bfsMin`, Algorithm (2). The values of `bfsMin` are sorted in increasing order across the x-axis and normalized for the 45 SPLOT models analyzed. Because `bfsMin` is an exhaustive search method, its execution time took the longest in almost all cases (only in one instance a random solution took slightly longer) ranging from 0.7 ms to 597.34 ms. In this latter case, the `maxComm` and `randComp` solutions took less than 1% time of their `bfsMin` counterpart.

6. RELATED WORK

Our previous work illustrated how safe composition could be used for detecting structural inconsistencies in multi-view UML models [22]. There has also been work on checking consistency in other advanced modularization approaches, such as aspects [20]; however, to the best of our knowledge, they do not address consistency in the presence of variability.

The work by Janota and Botterweck has a similar objective, detecting inconsistencies between the variability expressed in feature models and so-called architectural models (e.g. component models) [19]. An important difference with our work is that a feature can be realized by multiple models and symmetrically a model can implement more than one feature; in other words, they do not aim at modularizing features. Their approach works by finding implicit constraints in the architectural models and use them to enhance the process of product configuration in an iterative manner.

The work by Atkins et al. proposes Orthographic Software Modeling (OSM) as an approach that aims at providing multi-view consistency management across multiple dimensions such as variants to capture SPL variability or abstraction to represent ideas of Model-Driven Development [3]. A key characteristic of this work is the Single Underlying Model (SUM) that contains all the information from all the views. Though keeping all views consistent is a goal of OSM, it is unclear for us if this consistency also includes checking all possible product configurations defined with a feature model as safe composition does.

The work presented in this paper focuses on structural properties of SPL. However, there is an increasing number of works in verifying behavioral properties. For instance, Classen et al. propose an extension to transition systems that reifies the concept of features [11].

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented a first step towards fixing inconsistencies in models that have variability. We raised the issue of where fixes should be placed. We analyzed two approaches, a random approach and a heuristic approach based on pairwise commonality values, in terms of the size of their fixing sets and their execution time. We used a BFS-based approach as a baseline for our comparison. Overall, the heuristic approach does find shorter fixing sets than the random one, but comes with the overhead of computation of the pairwise commonality values which may undermine its overall performance. Additionally, the heuristic approach yields results that in some cases are far from the optimal found by the exhaustive BFS-based approach.

There are several issues that we plan to address as part of our future work:

- Consistency rules that involve more than one feature in the requiring or required elements, in other words, i.e. eliminate our Assumption 1.
- Fixing multiple inconsistency rule instances so that fixing one may cause inconsistencies in another instance of the same or other rule, i.e. eliminate our Assumption 2.
- Rules that also indicate composition conflicts, meaning that two sets of elements should not be present together for certain feature configurations [22].
- More complex fixing operations, that do not only consider adding fixes to features but deleting or moving the required elements across features.

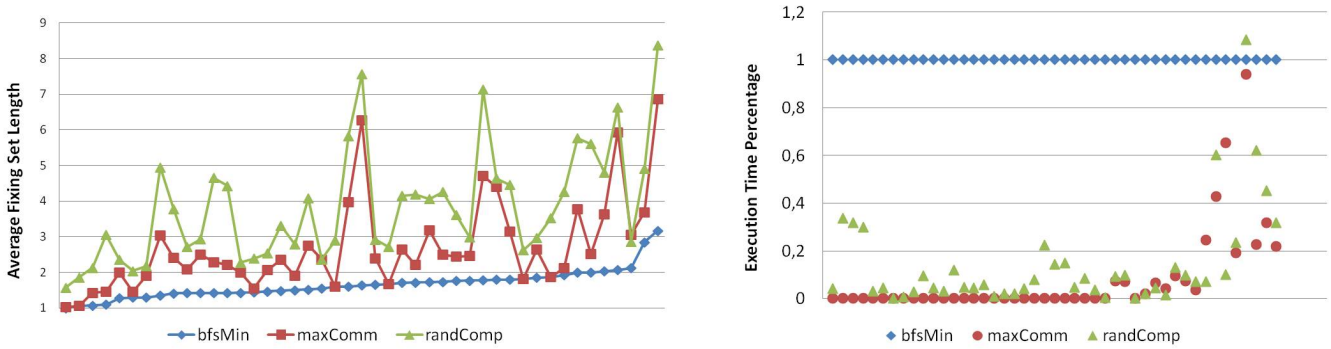

Figure 6: Fixing Sets average length and computation time

- Combination of heuristics and random searches with the goal of approximating to the optimal but without the performance penalty.

To address these open issues, we are currently exploring heuristic problem solving [25] and search-based software engineering techniques [24]. They reformulate the open issues like ours as optimization problems, for which there might not be optimal or tractable solutions, but instead good-enough solutions based on some optimization or fitness criteria.

Acknowledgments. We thank Alexander Nöhrer for his help with PicoSAT solver, Pablo Trinidad and David Benavides for their support with the FAMA tool suite. This research was partially funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965.

8. REFERENCES

[1] FAMA Tool Suite, 2010. http://www.isa.us.es/fama/.

[2] Software Product Line Online Tools(SPLOT), Accesed July 2011. http://www.splot-research.org/.

[3] C. Atkinson, D. Stoll, and P. Bostan. Supporting view-based development through orthographic software modeling. In *The 4th International Conference on Evaluation of Novel Approaches to Software Engineering ENASE*, 2009.

[4] D. Batory. AHEAD Tool Suite, 2010. http://www.cs.utexas.edu/users/schwartz/ATS.html.

[5] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[6] D. S. Batory. Using modern mathematics as an fosd modeling language. In Smaragdakis and Siek [27], pages 35–44.

[7] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

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

[9] X. Blanc, A. Mougenot, I. Mounier, and T. Mens. Incremental detection of model inconsistencies based on model operations. In P. van Eck, J. Gordijn, and R. Wieringa, editors, *CAiSE*, volume 5565 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2009.

[10] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 511–520. ACM, 2008.

[11] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking <u>lots</u> of systems: efficient verification of temporal properties in software product lines. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *ICSE (1)*, pages 335–344. ACM, 2010.

[12] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[13] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE*, pages 211–220. ACM, 2006.

[14] B. Delaware, W. R. Cook, and D. S. Batory. Fitting the pieces together: a machine-checked model of safe composition. In van Vliet and Issarny [29], pages 243–252.

[15] A. Egyed. Instant consistency checking for the uml. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 381–390. ACM, 2006.

[16] A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in uml design models. In *ASE*, pages 99–108. IEEE, 2008.

[17] I. Groher and M. Völter. Aspect-oriented model-driven software product line engineering. *T. Aspect-Oriented Software Development VI*, 6:111–152, 2009.

[18] M. Huth and M. Ryan. *Logic in Computer Science. Modelling and Reasoning about systems*. Cambridge University Press, 2004.

[19] M. Janota and G. Botterweck. Formal approach to integrating feature and architecture models. In J. L. Fiadeiro and P. Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2008.

[20] J. Kienzle, W. A. Abed, and J. Klein. Aspect-oriented multi-view modeling. In K. J. Sullivan, editor, *AOSD*, pages 87–98. ACM, 2009.

[21] R. E. Lopez-Herrejon and D. S. Batory. A standard problem for evaluating product-line methodologies. In *GCSE*, pages 10–24, 2001.

[22] R. E. Lopez-Herrejon and A. Egyed. Detecting inconsistencies in multi-view models with variability. In T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, editors, *ECMFA*, volume 6138 of *Lecture Notes in Computer*

Science, pages 217–232. Springer, 2010.

[23] F. J. Lucas, F. Molina, and J. A. T. Álvarez. A systematic review of uml model consistency management. *Information & Software Technology*, 51(12):1631–1645, 2009.

[24] S. A. M. Mark Harman and Y. Zhang. Technical report, King's College London, TR-09-03, April, Title = Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications, Year = 2009.

[25] Z. Michalewicz. *How to Solve It: Modern Heuristics*. Springer, 2010.

[26] OMG. Uml infrastructure specification v2.2. 2009.

[27] Y. Smaragdakis and J. G. Siek, editors. *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*. ACM, 2008.

[28] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In C. Consel and J. L. Lawall, editors, *GPCE*, pages 95–104. ACM, 2007.

[29] H. van Vliet and V. Issarny, editors. *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. ACM, 2009.

[30] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In van Vliet and Issarny [29], pages 315–324.