Journal of Universal Computer Science, vol. 17, no. 5 (2011), 684-711
submitted: 8/2/10, accepted: 4/11/10, appeared: 1/3/11 © J.UCS

Automatically Checking Feature Model Refactorings

Rohit Gheyi
(Department of Computing and Systems
Federal University of Campina Grande, Brazil
rohit@dsc.ufcg.edu.br)

Tiago Massoni
(Department of Computing and Systems
Federal University of Campina Grande, Brazil
massoni@dsc.ufcg.edu.br)

Paulo Borba
(Informatics Center
Federal University of Pernambuco, Brazil
phmb@cin.ufpe.br)

Abstract: A feature model (FM) defines the valid combinations of features, whose combinations correspond to a program in a Software Product Line (SPL). FMs may evolve, for instance, during refactoring activities. Developers may use a catalog of refactorings as support. However, the catalog is incomplete in principle. Additionally, it is non-trivial to propose correct refactorings. To our knowledge, no previous analysis technique for FMs is used for checking properties of general FM refactorings (a transformation that can be applied to a number of FMs) containing a representative number of features. We propose an efficient encoding of FMs in the Alloy formal specification language. Based on this encoding, we show how the Alloy Analyzer tool, which performs analysis on Alloy models, can be used to automatically check whether encoded general and specific FM refactorings are correct. Our approach can analyze general transformations automatically to a significant scale in a few seconds. In order to evaluate the analysis performance of our encoding, we evaluated in automatically generated FMs ranging from 500 to 2,000 features. Furthermore, we analyze the soundness of general transformations.

Key Words: feature models, refactoring, Alloy
Category: D.2.2, D.2.7, D.2.13

1 Introduction

Feature Models (FM) [Czarnecki and Eisenecker 2000] specify configurability constraints for a Software Product Line (SPL) [Clements and Northrop 2001]. A FM represents the common and variable features of a SPL and their dependencies. In order to include new products, a SPL may evolve. Evolution efforts can be supported by *refactorings* [Fowler 1999]. An extended definition for refactoring SPL – program and associated FMs – has been defined [Alves et al. 2006].

A SPL refactoring is performed by program and FM refactorings. A FM refactoring must improve the quality of a FM by maintaining or increasing its configurability. To help developers in refactoring SPL, a catalog containing a number of general FM refactorings was proposed [Alves et al. 2006], preserving configurability. This catalog is proved to be sound, complete and minimal. However, catalogs of FM refactorings that increase configurability are incomplete in principle. Therefore sometimes developers may have to refactor FMs based on *ad hoc* reasoning, an error-prone activity. In order to address this problem, refactoring designers must propose new transformations.

It is non-trivial to propose correct refactorings. Checking soundness by *ad hoc* reasoning or in a theorem prover [Gheyi et al. 2006a] is hard and time-consuming, requiring extra expertise. As shown by Batory [Batory 2005], FMs can be translated into propositional formula, in this translation enables analysis with existing logic-based tools, such as SATisfiability Problem (SAT) solvers. Accordingly, Benavides et al. [Benavides et al. 2006a] survey a number of useful analysis on FMs. Still, these tools do not focus on refactoring analysis.

This article extends our earlier work [Alves et al. 2006] by proposing an efficient encoding of FMs in Alloy [Jackson 2006], which is a formal specification language. We consider FMs freely containing propositional formulas. Based on this encoding, the Alloy Analyzer [Jackson et al. 2000], which is a tool used to perform analysis on Alloy models, helps refactoring designers to automatically and efficiently check whether FM refactorings are sound. The Alloy Analyzer is backed by SAT solving techniques as well. Moreover, this approach helps developers when refactoring SPLs by automatically verifying whether FM transformations to specific programs constitute a refactoring. If the transformation is not a refactoring, the tool generates a counterexample. The main contributions of this article are the following:

- An efficient encoding of FMs in Alloy (Section 3);
- An automatic method for checking whether a general or specific FM transformation is a refactoring (Section 4).

We evaluate our approach by checking the correctness of 19 general proposed refactorings [Alves et al. 2006]. Moreover, we used the benchmark proposed by Mendonça et al. [Mendonca et al. 2008] to evaluate the analysis performance of our encoding. We used their automatically-generated FMs ranging from 500 to 2,000 features each, and up to ECR = 30% (the ratio of the number of variables in the extra constraints to the number of variables in the feature tree [Mendonca et al. 2008]). When checking satisfiability of these FMs, our analysis took a few seconds. Mendonça et. al, which propose an efficient compilation technique for BDDs, performed the same experiment. They could analyze two out of ten FMs containing 2,000 features (ECR=30%) in at most 94 seconds. They had memory overflow issues, which did not occur in our approach, when analyzing the other 8 FMs. The analysis performance and the number of features involved in our approach seems to be reasonable, since most of FMs in practice contains less than thousands of features.

The remainder of this document is organized as follows. We describe the state of the art and discuss issues in FM refactorings in Section 2. We overview Alloy and present an encoding of FMs in Alloy in Section 3. Section 4 explains how to specify and analyze FM refactorings using our encoding. Finally, we show the related work and conclusions in Sections 5 and 6, respectively.

2 Overview

In this section, we give a brief overview of FMs (Section 2.1) and SPL refactoring (Section 2.2). Finally, Section 2.3 presents some issues in FM refactorings.

2.1 Feature models

A FM represents the common and variable features of a SPL and the dependencies between them [Czarnecki and Eisenecker 2000, Kang et al. 1990]. A feature diagram is a tree-like graphical representation of a feature model. Relationships between a parent feature and its child features (also regarded as subfeatures) may be *Optional* (represented by an unfilled circle), *Mandatory* (represented by an filled circle), *Or* – one or more must be selected (represented by a filled triangle), and *Alternative* – exactly one subfeature must be selected (represented by a unfilled triangle). A FM may also include cardinality constraints [Czarnecki et al. 2005], but they are not considered here. Figure 1 depicts these relationships graphically.

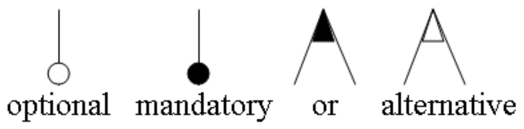


Figure 1: Feature Diagram Notations

Besides these relationships, FMs may include propositional logic formulas about features. For instance, the formula $earphone \Leftrightarrow mp3$ states that the feature *earphone* is selected if and only if the feature $mp3$ is selected.

Figure 2 depicts a simplified FM for a mobile phone. A mobile phone may have an earphone. Moreover, it may have at least an mp3 player or a digital camera. Finally, a mobile phone has an earphone if and only if it has a mp3 player. So, the FM has four features (*mobilephone*, *earphone*, *mp*3 and *camera*), one formula ($earphone \Leftrightarrow mp3$) and two relations: an optional relation between *mobilephone* and *earphone*, and an *or* feature relation between *mobilephone*, *mp*3 and *camera*.

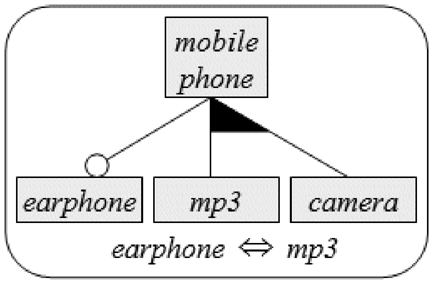


Figure 2: Feature Model Example

Although there are a number of proposals for defining meaning for relationships in FMs (see work by Schobbens et al. [Schobbens et al. 2007] for a survey), the most usual semantics for a FM is given by the set of its valid configurations (possibly instantiated software products). A valid configuration contains a set of feature names; if *valid*, it satisfies all constraints (relations and formulas) of the model. For example, the configuration ($\{mobilephone, camera\}$) is valid for the model in Figure 2 representing a mobile phone only with camera. However, the configuration ($\{mobilephone, earphone\}$) is invalid because the or feature relation between *mobilephone*, *mp*3 and *camera* states that whenever *mobilephone* is selected, at least *mp*3 or *camera* must be selected. Also, the formula $earphone \Leftrightarrow mp3$ is violated.

2.2 Refactoring Product Lines

In order to motivate the need for FM refactorings, in this section, we explain issues that need to be addressed when considering refactoring in the SPL context. Then we present an extended definition of refactoring for such context.

The term refactoring was coined by Opdyke in his thesis [Opdyke 1992]. He proposed refactorings as behavior-preserving program transformations; initially the definition supports the iterative design of object-oriented application frameworks [Opdyke 1992]. The cornerstone of his definition is that refactorings must maintain correct compilation and observable behavior. In practice, behavior preservation is evaluated by successive compilation and tests. Opdyke's work and many of the later refactoring definitions apply to frameworks (heavily used in current SPL development) and often introduce variation points.

Nevertheless, as *program transformations*, they do not handle configurability issues (only tackled at FM level). In practice, a decrement in SPL configurability while refactoring should not occur. If a set of products represented by A_1 are correctly refactored into a set of products represented by A_2, following traditional refactoring steps (compilation and unit tests), still it is not guaranteed that configurability is preserved. We must certify that FM_2 (corresponding to A_2) preserves the same possible configurations as FM_1 (corresponding to A_1). Since the configurability is described by the FM, such model should also be considered during SPL refactoring.

Our previous work with others extends the definition of refactoring for dealing with SPLs [Alves et al. 2006]:

Definition 1 *SPL refactoring is a change made to the structure of a SPL in order to improve (preserve or increase) its configurability, without changing the observable behavior of its original products.*

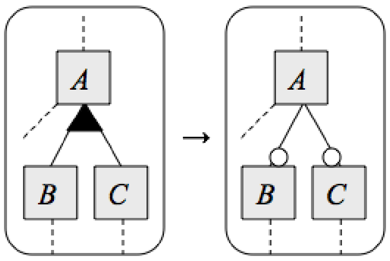
In our definition, maintaining or increasing configurations of a SPL is a desirable quality. Alves et al. [Alves et al. 2007] present some program refactorings for SPL, which are implemented using Aspect-Oriented Programming. Some of these ideas have been implemented in a supporting tool [Soares et al. 2008]. So, a SPL refactoring thus can be seen as a program refactoring plus a FM refactoring, which is defined next [Alves et al. 2006]:

Definition 2 *A FM refactoring is a transformation that improves the quality of a FM by maintaining or increasing its configurability. So, the resulting FM contains all valid configurations of the initial FM, but may contain more.*

Developers can apply general FM refactorings based on template matching [Alves et al. 2006] to check whether a transformation is correct. Each *general refactoring* consists of two *templates* (patterns) of FMs, on the left-hand (LHS) and right-hand (RHS) sides. A refactoring can be applied whenever the FM matches the template. A matching is an assignment of all meta-features occurring in the LHS template to concrete values. Any element omitted by the templates remains unchanged, thus refactoring templates only show differences between FMs. Moreover, a dashed line on top of a feature indicates that this feature may have a parent feature. A dashed line below a feature indicates that this feature may have additional subfeatures.

As an example, Refactoring 1 is a general transformation (we can apply it to a number of FMs that match its template) that transforms an or relationship into two optional features. The transformation increases the configurability of the resulting FM by allowing the additional configuration {A}. A, B, and C are meta-features.

Refactoring 1 ⟨*change or to option*⟩



For instance, the FM depicted in Figure 2 matches the LHS FM template of Refactoring 1. The meta-features A, B and C are matched with *mobilephone*, *mp*3 and *camera*, respectively.

Our earlier work [Alves et al. 2006] also presents a different kind of transformation: *merging*. This transformation combines a number (greater than one) of FMs into a single FM, different from a FM refactoring which relates a single FM to another one. Segura et al. [Segura et al. 2008] also present a number of merging transformations. In this work, we focus on FM refactorings. But the results of this work can be similarly applied to merging transformations.

2.3 Issues in Feature Model Refactorings

In this article, we focus on issues related to FM refactorings. A catalog of FM refactorings (preserving or increasing configurability) was proposed [Alves et al. 2006, Segura et al. 2007], helping developers evolve FMs. The catalog of FM refactorings that maintain configurability (called B-refactorings [Alves et al. 2006, Gheyi et al. 2008] – the original and refactored FMs contain the same configurations) is proved to be sound, complete and minimal. However, the catalog of refactorings that *increase configurability* (the refactored FM contains at least the same configurations of the original one) is not proved to be complete.

Since the previously proposed catalog [Alves et al. 2006, Segura et al. 2007, van Deursen and Klint 2002] is incomplete, FM refactoring may demand intuition to check whether a transformation that they wish to carry out is indeed a sound refactoring that increases configurability. However, it is difficult, time-consuming and error-prone to directly reason about FM semantics to check configurability.

In order to solve this problem, more general refactorings (Refactoring 1) have to be defined. However, there are no strict guidelines to follow when defining sound refactorings. FM refactorings then must be proved manually or with the help of a theorem prover [Gheyi et al. 2006a]. Nevertheless, as it is widely known, theorem proving is a demanding task and requires sophisticated expertise. Interactive proofs for proposed FM refactorings may be time consuming [Gheyi et al. 2006a].

Additional issue with proofs rises from that fact that failure to prove FM transformations is meaningless for stating that they are not refactorings. In order to show this non-conformance, refactoring designers must discover a configuration (counterexample) that belongs to the initial FM but does not belong to the resulting FM. It is not straightforward to find counterexamples in FMs with hundreds of features that may represent thousands of products. Therefore, refactoring designers may waste time trying to prove transformations that are not refactorings. In this context, the current approach for proving FM refactorings is not practical. Refactoring designers need a more effective way to increase the catalog of sound FM refactorings.

3 Feature Models in Alloy

In this section, we specify an efficient encoding of FMs using Alloy. We choose Alloy due to its tool support, which can perform analysis, because it was appropriate of easy to use for formulating our particular type of problems. Moreover, it contains a functionality (the unsat core) that helps us debugging inconsistent models [Torlak et al. 2008].

First we give a brief overview of Alloy (Section 3.1). Section 3.2 presents our encoding that is intended to solve the problems presented in Section 2.3. The approach is generalized in Section 3.3. Finally, Section 3.4 shows how the encoding can be used to perform analysis on FMs.

3.1 Alloy Overview

An Alloy model or specification is a sequence of *paragraphs* of two kinds: signatures that are used for defining new types, and constraints, such as facts. Each signature denotes a set of objects (similar to an UML class), which are associated to other objects by relations declared in the signatures. A signature paragraph introduces a type and a collection of relations, called fields, along with their types and other constraints on their included values.

The Alloy Analyzer tool [Jackson et al. 2000] allows us to perform analysis on an Alloy specification in order, for instance, to check whether some properties are deducible from the model for a pre-defined scope. A scope defines the maximum number of objects allowed for each signature during analysis. The tool assigns a bound to the number of objects of each type. The tool automatically searches all possible situations up to a given scope. As you increase the scope, the space of cases to consider grows dramatically. Even a small scope usually defines a huge space [Jackson 2006]. In the default scope of 3, for example, which assigns a bound of three to each signature, each binary relation contributes 9 elements (since each three elements of the domain may or may not be associated with each three elements of the range) — that is, a factor of 512 combinations (the number of all possible subsets with 9 elements). So a tiny model with only four relations may have a space of over a billion cases [Jackson 2006].

The simulations performed by the Alloy Analyzer tool are sound and complete up to a given scope. If there is some instance that contradicts an assertion up to a given scope, the tool shows a counterexample. However, if the tool does not find any counterexample, we only know that the property holds on that scope. We cannot conclude that the formulas declared in the assertion are valid for a greater scope since the tool is not a theorem prover. By increasing the scope, however, we can gain greater confidence. In some specific domains, we exactly know the number of instances involved of each signature. In this situation, we can perform a complete analysis (see Section 4).

3.2 Encoding

A FM encompasses a set of feature names. In our encoding, there are two signatures (*FM* and *Name*) and one relation (*features*) representing all elements of FMs.

```
sig FM {
  features: set Name
}
sig Name {}
```

The **set** qualifier specifies that the relation *features* associates each element in *FM* to a set of elements in *Name*.

As described before, a configuration contains a set of feature names selected. We might have declared the signature *Configuration* with one relation to a set of selected feature names. In this case, we need to define a scope for this signature. However, we are constrained by the performance of the Alloy Analyzer regarding number of signatures, so we declare as few signatures as possible. The Alloy Analyzer searches all possible combinations for a solution by giving all possible values to all signatures up to a given scope. By decreasing the number of signatures, it decreases the number of combinations, hence increasing performance. In our theory, since it only contains two signatures, we only need to define the scope of two signatures. So, a variable (*conf*) is then defined for the following predicates, typed as a set of feature names. In this way, we do not declare a new signature. The configuration values are implicitly defined by subsets of the values given to names. It will generate the subsets of names on demand different from the previous approach (declaring a signature), in which it is always generated all possible configurations, hence decreasing performance.

Relationships between features (Figure 1) are declared as Alloy formulas – one *predicate* is given for each FM relation. In Alloy, ***predicates*** (**pred**) are used to package reusable formulas. Given arbitrary feature A and its subfeature (child) B, next we specify the optional and mandatory relationships between A and B in predicates. We also state a predicate for the root (*root*) of a FM. For a given configuration (*conf*), the root feature must be included.

```
pred optional[A,B:Name, conf:set Name] {
  B in conf ⇒ A in conf
}
pred mandatory[A,B:Name, conf:set Name]{
  A in conf ⇔ B in conf
}
pred root[A:Name, conf:set Name] {
  A in conf
}
```

The **in** keyword denotes the subset operator. Suppose that A is related to a number of subfeatures *children*. Next we specify the or and alternative relationships in predicates.

```
pred orFeature[A:Name, children:set Name, conf:set Name] {
  A in conf ⇔ some c:children | c in conf
  #children > 1
}
pred alternative[A:Name, children:set Name, conf:set Name] {
  orFeature[A,children,conf]
  #children & conf < = 1
}
```

The **some** keyword represents the existential quantifier. The **#** and **&** operators denote the cardinality of a set and the intersection set operator, respectively. A FM may also contain a set of propositional formulas. A straightforward encoding for these formulas is to define an additional relation in *FM* to link each FM to a set of formulas, being possibly represented by a *Formula* signature. In such encoding method, a signature hierarchy representing all kinds of propositional formulas (conjunctions, disjunctions) must be specified. Moreover, a recursive predicate that checks whether a configuration satisfies a formula is needed. Since Alloy 4 does not offer explicit support for recursive predicates, the resulting specification is not very readable. In addition, as mentioned before, additional signatures in general decreases the Alloy Analyzer performance. Therefore, we devised an encoding method that abstracts formulas' syntax; they are encoded based on their semantics, similarly to the relation predicates previously showed in this section. Notice that all predicates representing FM relationships contain the variable *conf*.

3.2.1 Example

Suppose that we would like to represent the FM in Figure 2 using the encoding presented in Section 3.2. In Alloy, one signature can extend another, establishing that the extended signature (***subsignature***) is a subset of the parent signature. Firstly, we declare its elements; a singleton (`one`) subsignature, which has exactly one object, for each FM element. The FM in Figure 2 is represented by *M*, which extends *FM*, and presents 4 features. A singleton signature is declared for each feature name. Finally we state *M*'s features in a fact (**fact**), which packages formulas that always hold, such as invariants about the elements.

```
one sig M extends FM {}
one sig mobilePhone, earphone, mp3, camera extends Name {}
fact MFeatures {
  M·features = mobilePhone + earphone + mp3 + camera
}
```

The + operator denotes the set union operator. Our main goal is to reason whether a transformation preserves or increases FM configurations. For that, FM semantics must be specified in Alloy. One approach is to declare an Alloy function yielding a set of valid configurations for a FM. By our concern in improving analysis performance, we cannot declare a semantics function for all FMs, which could be very inefficient. We then specified a *semantics* predicate for each FM. Part of this predicate is fixed for all FMs. The other part depends on its relationships and formulas. This encoding is systematic, straightforward for being included into tool support. Next, we explain the encoding through an example, later generalizing our approach.

For each FM, a predicate is defined, containing all FM formulas directly translated to their semantics function. Using this approach, there is no predicate checking whether a configuration satisfies a formula. The immutable part of the *semantics* predicate introduce the following constraints: every configuration includes a subset of FMs names, and the root must always be included, as declared next. We call them *implicit constraints*.

```
pred semanticsM[conf: set Name] {
  conf in M·features
  root[mobilePhone,conf]
```

Then all relationships of the FM are declared in terms of the predicates presented in Section 3.2.

optional[mobilePhone,earphone,conf]
orFeature[mobilePhone,earphone+ camera,conf]

After specifying all relationships and implicit constraints, now we specify all explicit propositional formulas from the model. Figure 2 presents one formula ($earphone \Leftrightarrow mp3$). Each formula's operator is directly translated to an equivalent one in Alloy. Every occurrence of a feature name is appended with `in` *conf*. Any propositional formula can be directly translated to Alloy. Next we present how to encode the previous formula.

earphone **in** conf ⇔ mp3 **in** conf
}

3.3 Generalization

In order to systematically specify a FM into Alloy using our encoding, the following steps must be taken:

- create a singleton subsignature for each feature extending from *Name*;
- specify the semantics predicate containing the relationships (reusing the encoding predicates) and formulas (using Alloy operators) in the FM.

In summary, the semantics predicate for any FM must include the two mandatory implicit constraints and the translation of all relationships and formulas declared in the FM. Relationships are specified using the predicates in our encoding, and formulas are translated to equivalent ones in Alloy by appending `in` *conf* to each name in the formula. This translation from FM to Alloy is systematic and can be easily implemented by a tool.

3.4 Analysis

Based on the previous encoding, we can perform automatic analysis on FMs using the Alloy Analyzer. Figure 2 has exactly one FM and 4 feature names. Since it is known the exact number of objects for all signatures (*FM* and *Name*) in our encoding, we can perform a *complete analysis* using the Alloy Analyzer in the FM of Figure 2.

We apply the two analyses in this section only for better illustrating our approach. Analysis on a single FM has been tackled by several research contributions, such as Batory and Benavides et al. [Batory 2005, Benavides et al. 2006a], so to this respect our approach is not novel. Trinidad et al. and White et al. [Trinidad et al. 2008a, White et al. 2008] show how to debug inconsistent FMs and configurations. The analyses described in this section can provide an additional benefit: the *Unsat Core* functionality from the Alloy Analyzer, which may offer help to avoid extensive debugging when searching for errors in FMs.

Verifying a Configuration

We can use Alloy's predicates to check whether a specific configuration is valid for a FM. For example, the following predicate states whether selecting *mobilePhone* and *earphone* is a valid configuration for *M*.

```
pred validConfig [] {
  semanticsM[mobilePhone+ earphone]
}
run validConfig for 1 FM, 4 Name
```

The *run* analysis command must specify a scope for all signatures declared. Our encoding contains 2 signatures. The previous fragment declares the *run* command for one FM (we are analyzing only one FM) and for 4 names (the FM encoded contains 4 features). Performing analysis on the previous predicate, with the *run* command, no valid configuration is yielded. It means that choosing *mobilePhone* and *earphone* does not encompass a valid configuration for *M*.

In these situations, the Alloy Analyzer tool contains the unsat core functionality, which pinpoints an irreducible unsatisfiable core of a declarative specification. It highlights constraints that may be the cause of inconsistency. Running the previous predicate for the previous configuration with additional *mp*3 yields validity. Therefore, this is a valid configuration for *M*.

Finding a Valid Configuration

Furthermore, we can use Alloy Analyzer to automatically provide configurations for a given FM. For instance, we can perform analysis to show valid configurations of *M* by running the *semanticsM* predicate with the run command. The Alloy Analyzer yields a solution.

```
run semanticsM for 1 FM, 4 Name
```

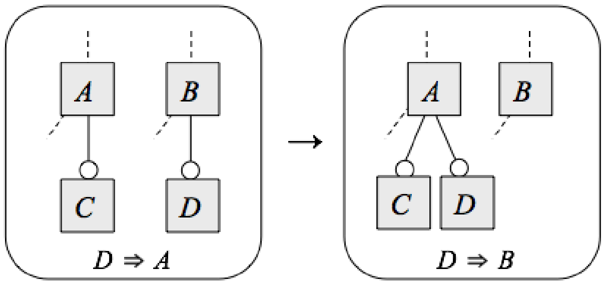
4 Checking Refactoring Soundness

In this section, we explain the main benefit of our encoding: checking whether general FM refactorings are sound – preserving or increasing configurability – up to a given scope (Section 4.1). Further, we describe alternative analysis purposes, such as verifying individual transformations (Section 4.2), followed by an account about performance of our encoding (Section 4.3).

4.1 Encoding General Refactorings

Suppose that a refactoring designer proposes Refactoring 2, which allows one to move an optional subfeature to a parent feature, as long as a specified formula holds. This formula is modified in the resulting FM. A, B, C and D represent meta-features.

Refactoring 2 ⟨*move optional feature*⟩



In order to check soundness, the syntactic relationship between the LHS and RHS FMs must be specified with Alloy signatures and facts; this is done by specifying common and constrasting elements. Also, the semantics of each FM is specified using an Alloy predicate. Only some elements of the specification, which are called *hot spots*, must be specified.

Abstract Syntax

Refactoring 2 defines two FMs. The LHS and RHS FMs are represented as *fm1* and *fm2*, respectively. They include four meta-features: A, B, C and D. Singleton (**one**) subsignatures represent both FMs and all names.

```
one sig fm1,fm2 extends FM {}
one sig A,B,C,D extends Name {}
```

All meta-features belong to both FMs. We specify that *A, B, C* and *D* belong to *fm1*. Notice that *fm1* may contain more features not depicted in the transformations; unspecified elements in the meta-FMs remain the same. Therefore, in this case, both models have the same features.

```
fact SyntacticRelationship {
  A+ B+ C+ D in fm1·features
  fm1·features = fm2·features
}
```

Semantics

Regarding semantics, each FM has an implicit constraint: a configuration must include a subset of the features in the FM. Notice that *fm1* declares two optional relationships between A and C, and B and D, whereas *fm2* contains two optional relationships between A and C, and A and D. The explicit constraints must also be specified ($D \Rightarrow A$ and $D \Rightarrow B$). The previous constraints are specified in following predicates.

```
pred semanticsM1[config: set Name] {
  config in fm1·features
  optional[A,C,config]
  optional[B,D,config]
  D in config ⇒ A in config
}
pred semanticsM2[config: set Name] {
  config in fm2·features
  optional[A,C,config]
  optional[A,D,config]
  D in config ⇒ B in config
}
```

As explained in Section 3, the semantics predicate of a FM must include the root implicit fact and a number of formulas (other than the two formulas specified before). Notice that they are not specified in *semanticsM1* and *semanticsM2*; in fact, when checking FM refactorings, we do not need to include a formula that appears in both FMs (this is explained in Section 4.1.1).

Analysis

The analysis is accomplished by means of the following assertion, which tests the definition of a FM refactoring (Section 2.2). Assertions (**assert**) formula paragraphs that declare a set of questions about a model. The following assertion specifies whether all valid configurations of *fm1* are also valid configurations of *fm2*. Notice that we are checking a *meta-property*, which is a valid property for a number of FMs matching the templates.

```
assert refactoring {
  all config:set Name | semanticsM1[config] ⇒ semanticsM2[config]
}
check refactoring for 2 FM, 40000 Name
```

Notice that only the scope of *FM* and *Name* signatures must be specified. Each refactoring defines two *FM*s, representing the LHS and RHS models; therefore, the scope of two (2) for *FM* is defined. We do not know the number of features of general FM refactorings, since a FM containing any number of features can match the templates. So, in each refactoring the maximum number x of features considered in the analysis must be stipulated. If the Alloy Analyzer does not give any counterexample up to a given scope x, we can apply the FM refactoring to any FM containing less than or equal to x features.

The Alloy Analyzer can check whether the previous assertion is valid (deducible from the model constraints) up to a given scope. This property is checked in all possible situations containing at most two feature models and 40,000 feature names. For Refactoring 2 the Alloy Analyzer does not find any counterexample, with a scope of 40,000 features. Consequently, the refactoring is sound for FMs containing less than 40,000 features and matching the *fm1* template. As in practice FMs with such set of features are rare, Refactoring 2 is safe for general application.

The previous analysis took less than a second to be accomplished. We also checked all 19 FM refactorings proposed in our previous work [Alves et al. 2006, Gheyi et al. 2008]. We checked them using a scope of 10,000 in less than two hours. None of them yields a counterexample. Our analyses have been performed on an Intel Centrino Duo system with a 1.6 GHz processor and 1 GB RAM and running Windows XP.

4.1.1 Unchanged Formulas

The FMs in Refactoring 2 contain the same relationships and formulas, except for optional relationships and the replaced formula. Relationships are represented by formulas with the predicates from Section 3.2. Thus a set of formulas (relationships and explicit formulas) *forms* in both FMs are not depicted in Refactoring 2. All elements that are not mentioned in a refactoring remain unchanged in both FMs. So, strictly, both FMs encodings must consider *forms*. In order to specify a complete semantic definition, refactoring designers must include *forms* in the *semantics* predicate for Refactoring 2, as declared next.

```
pred semanticsM1[config: set Name] {
  config in fm1·features
  optional[A,C,config]
  optional[B,D,config]
  D in config ⇒ A in config
  forms
}
pred semanticsM2[config: set Name] {
  config in fm2·features
  optional[A,C,config]
  optional[A,D,config]
```

```
    D in config ⇒ B in config
    forms
  }
```

Nevertheless, we omitted *forms* in the encoding presented in Section 4.1. In fact, we do not need to specify *forms* since it is included in both FMs. The definition of FM refactoring states that all valid configurations of the LHS FM must be valid configurations of the RHS FM. If a configuration *config* satisfies the semantics of the LHS FM, it must satisfy *forms*. Since it is satisfied in the LHS FM, it is also satisfied in the RHS FM. So, they must focus only on the differences between the two FMs. The same argument explains why refactoring designers do not need to specify the implicit constraint on the *root feature*. Both FMs present the same root, hence the same implicit constraint. In fact, refactoring designers do not need to specify any constraint that appears in both FMs of a refactoring template.

4.1.2 Unsound General Transformations

Refactoring designers may define transformations that are intended to be refactorings, but are unsound (decreasing configurability). In this context, our approach leads to *counterexamples* demonstrating the unsoundness of such transformations.

For instance, refactoring designers may assume that the transformation depicted in Figure 3 is a sound refactoring. Since the two pairs of meta-features (B,C) and (D,E) exclude each other, the designer may be compelled to express the FM as two alternative relationships.

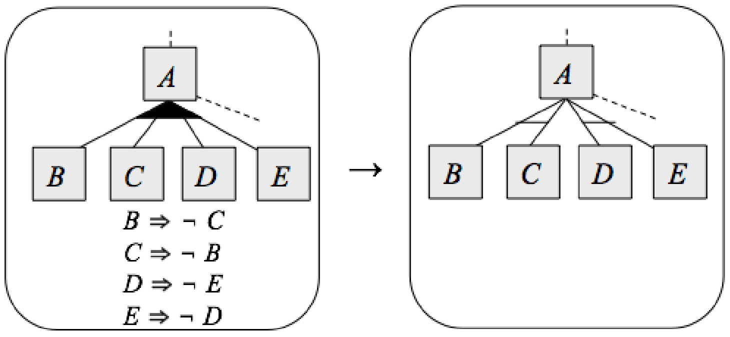


Figure 3: Unsound FM Refactoring

Following our approach, part of the model and the checking assertion are constructed as follows.

```
pred semanticsM1[config: set Name] {
  config in fm1·features
  orFeature[A,B+ C+ D+ E,config]
  B in config ⇒ C !in config
  C in config ⇒ B !in config
  D in config ⇒ E !in config
  E in config ⇒ D !in config
}
pred semanticsM2[config: set Name] {
  config in fm2·features
  alternative[A,B+ C,config]
  alternative[A,D+ E,config]
}
assert wrongRefactoring {
  all config:set Name | semanticsM1[config] ⇒ semanticsM2[config]
}
check wrongRefactoring for 2 FM, 5 Name
```

The Alloy Analyzer yields a configuration as counterexample for the assertion, with features A and B only. This configuration is valid for *fm1* of Figure 3, however invalid for *fm2*. Therefore, applying the transformation from left to right is not a refactoring since *fm2* does not contain all configurations of *fm1*. This counterexample is generated in less than a second. Checking the opposite transformation is a refactoring is straightforward; we create a similar assertion, and the tool does not give a counterexample for all possible FM containing up to 10,000 features. In this case, the analysis takes about 10 seconds to conclude.

One problem of our previous approach – proving refactorings in a theorem prover [Gheyi et al. 2006a]) – is that, in order to show that a transformation is not a refactoring, refactoring designers have to manually find a counterexample. In our approach, the Alloy Analyzer automatically yields counterexamples. The analyses performed by the Alloy Analyzer are fast, considering the number of features. In general, it takes a few seconds for FMs containing thousands of features. Previously, we had proved 11 general refactorings [Alves et al. 2006] using a theorem prover. This process is time consuming [Gheyi et al. 2006a]. Using the approach presented here, after manually translating all transformations to our encoding, we can automatically check those 11 refactorings using a scope of 10,000 features.

4.1.3 Generalization

In order to check a general FM refactoring, refactoring designers must concretize four *hot spots* from the encoding:

- extend *Name* with all meta-features in the refactoring;
- specify a syntactic relationship (the *features* relation) between source and target FMs;
- specify all relationships and formulas in a predicate defining semantics;
- specify the number of features for analysis.

Figure 4 depicts a general method to specify a general FM refactoring, using Refactoring 2. Only the hot spots require specification by refactoring designers. This approach is systematic; a tool can be built to automatically generate a specification in Alloy from a FM refactoring. The refactoring designers just need to specify the templates and the tool will detect whether it is a sound FM refactoring; Alloy knowledge is not required.

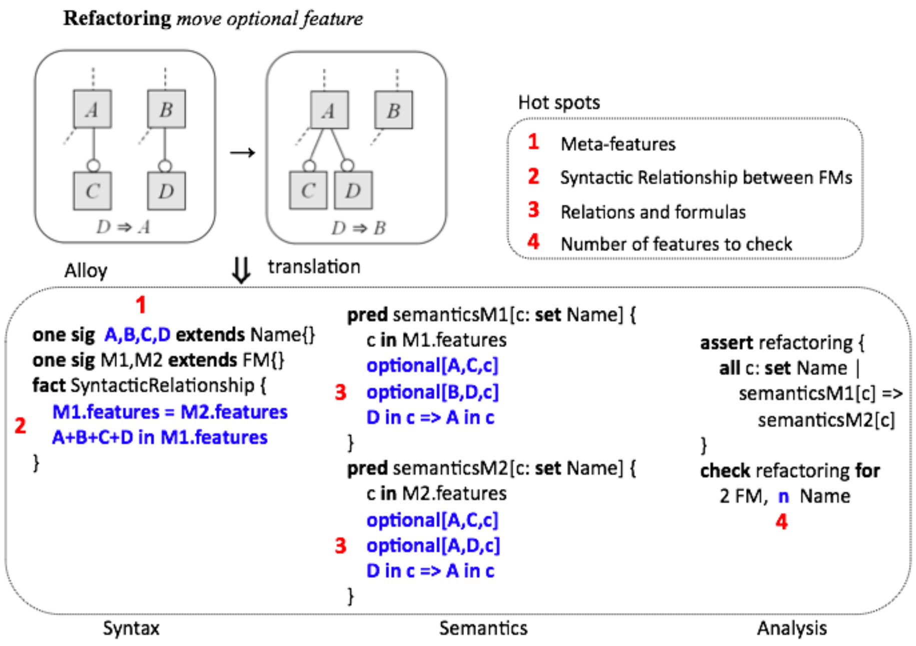
4.2 Analysis of Specific Transformations

Besides checking general transformations, our encoding can also be useful verifying specific transformations. Suppose we would like to prove the transformation depicted in Figure 5 is a refactoring.

In Section 3, we specify a FM that is similar to the LHS FM (*fm1*) in Figure 5. Next, we specify the RHS FM (*fm2*), following the guidelines explained in Section 4.1.3. This specification is simpler, as it involves concrete features.

```
pred semanticsM2[config:set Name] {
  config in M1·features
  root[mobilePhone,config]
  orFeature[mobilePhone,mp3+ camera,config]
  optional[mp3,earphone,config]
}
```

Checking whether *fm2* refactors *fm1* yields a counterexample (configuration) including *mobilePhone*, *camera* and *earphone*. This is a valid configuration for *fm1*, but it is invalid for *fm2*. Consequently, the transformation is not a refactoring, since it actually decreases the configurability. If we would like to check the transformation from *fm2* to *fm1*, it does not yield a counterexample. Since we exactly know the total number of features (4) involved in the transformation, the analysis performed by the Alloy Analyzer is *complete*. Therefore, we prove that *fm1* refactors *fm2*.



```
one sig A,B,C,D extends Name{}
one sig M1,M2 extends FM{}
fact SyntacticRelationship {
  M1.features = M2.features
  A+B+C+D in M1.features
}
```

```
pred semanticsM1[c: set Name] {
  c in M1.features
  optional[A,C,c]
  optional[B,D,c]
  D in c => A in c
}
pred semanticsM2[c: set Name] {
  c in M2.features
  optional[A,C,c]
  optional[A,D,c]
  D in c => A in c
}
```

```
assert refactoring {
  all c: set Name |
    semanticsM1[c] =>
      semanticsM2[c]
}
check refactoring for
  2 FM, n Name
```

Figure 4: Checking General FM Refactorings

Despite the small size of analyzed FMs, this approach can be useful when considering FMs containing a large number of features and big refactoring chains; in the latter, designers may waste time trying to find an appropriate sequence. If the analyzed transformation is not a refactoring, the automatically generated counterexample improves developers' understanding about the transformation, helping to identify the problem. The counterexample is not possible to yield by just using the catalog.

4.3 Performance Evaluation

We evaluated the analysis performance of our encoding in an experiment consisted on checking whether a FM contains a valid configuration. We used one real FM (the electronic shop FM containing 287 features and ECR=15%) and seven categories of automatically generated FMs containing up to 2,000 of features with 30% ECR. ECR is the ratio of the number of variables in the propositional formulas to the number of variables in the feature tree [Mendonca et al. 2008].

All analyses have been performed on an Intel Centrino Duo system with a 1.6 GHz processor and 1 GB RAM and running Windows XP.

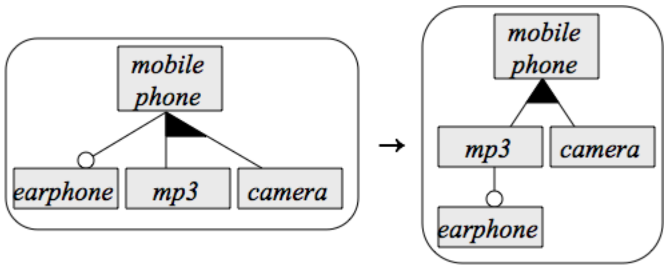
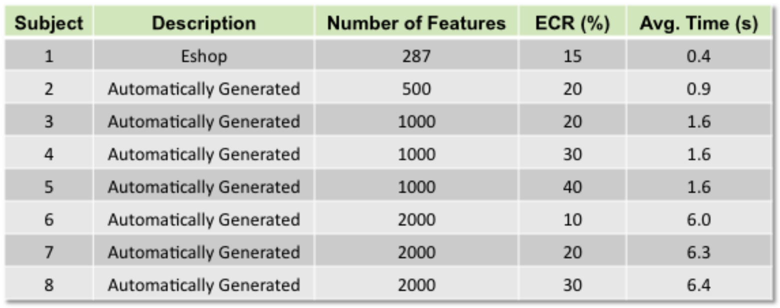


Figure 5: Proving a Specific Feature Model Refactoring



Subject	Description	Number of Features	ECR (%)	Avg. Time (s)
1	Eshop	287	15	0.4
2	Automatically Generated	500	20	0.9
3	Automatically Generated	1000	20	1.6
4	Automatically Generated	1000	30	1.6
5	Automatically Generated	1000	40	1.6
6	Automatically Generated	2000	10	6.0
7	Automatically Generated	2000	20	6.3
8	Automatically Generated	2000	30	6.4

Table 1: Summary of Analysis Performance

Table 1 summarizes the results. In each row, we show the time required by our approach to perform analysis. All analyses were performed in less than 7 seconds. The same analysis was performed by Mendonça et al. [Mendonca et al. 2008]. In all cases, our approach outperforms the mentioned work.

Mendonça et al. analyzed 10 FMs in Subject 8. In 80% of the cases, it gave memory overflow. In 20% of the FMs, the analysis took more than a minute. We evaluated each of the 10 FMs in less then 7 seconds without memory overflow. This result may indicate that our encoding scales.

5 Related Work

Batory [Batory 2005] integrates prior results to connect feature diagrams, grammars, and propositional formulas. This connection – similarly to ours – allows the use of SAT solvers to analyze FMs. He explains in details how to check whether: (1) a FM is inconsistent, and (2) a configuration is valid for a FM. Both properties can also be checked in the Alloy Analyzer, as explained in Section 3. He also explains how to carry out automatic FM instantiation. The user selects a number of features, then the tool yields a configuration suggestion for the remaining features that satisfies the FM semantics. Our approach can accomplish a similar result, by declaring a predicate very similar to the *semantics* predicate. The difference is that we add a formula, within the predicate, stating the features selected by the user. For example, performing analyses on the following predicate, the Alloy Analyzer yields valid configurations (if any) for the FM depicted in Figure 2 when the user selects *mobile Phone* and *earphone*. Besides these properties, the main goal of our work is to show how the Alloy Analyzer can be used for checking meta-properties (general refactorings). In some cases, theorem proving is even implied by the complete analyses.

```
pred partialConfiguration[conf: set Name] {
  semanticsM[conf]
  mobilePhone+ earphone in conf
}
```

Thüm et al. [Thum et al. 2009] present and evaluate an algorithm to determine the relationship between two feature models using satisfiability solvers. The relationship can be: specialization (reducing products), refactoring (maintaining products), generalization (including products) or none of these. The analysis in our work focus on product maintenance or addition, with Alloy and its Analyzer. From the performance evaluations, similar results can be observed. Our approach, likewise, can be applied to verify whether a transformation is a specialization; in this case, the verification aims at testing whether products in the refactored model are encompassed by the original model. Categorization of FM edits is a relevant kind of analysis; nevertheless Alloy assertions can go further in performing other sorts of user-defined analyses, so developers are not limited to categorizing edits. The generated counterexamples may also be useful for debugging. In addition, differently from the mentioned work, our work checks general transformations (with meta-properties), providing additional guidance to refactoring designers that establish commonly applied refactorings that do not need to be verified at each application.

Deursen and Klint [van Deursen and Klint 2002] propose a textual language for describing features. Their language is similar to the FM language considered in our work, but not encompassing formulas. The semantics, however, is equivalent to ours. Also, a set of 15 rules relating equivalent FMs are proposed, which are very similar to bidirectional refactorings [Alves et al. 2006]. Soundness is informally guaranteed, in contrast with our approach, which uses the Alloy Analyzer for increasing confidence on the result. We also give a tool support for checking whether a specific transformation is sound.

Benavides et al. [Benavides et al. 2005] propose an automatic way to analyze five FM properties, such as the number of possible configurations and an enumeration of all configurations, and check whether a FM is consistent. They present a mapping to transform an extended FM into a Constraint Satisfaction Problem (CSP) in order to formalize extended FMs using constraint programming. Using our approach, those five properties can also be checked. Their idea of filters is equivalent to formulas. In our encoding, we give flexibility to the user to write and check any kind of assertion based on our encoding. Moreover, Alloy analysis is efficient with respect to the number of features. The tool also has support for dealing with inconsistent models (Unsat Core). Our theory has a limited support for integer expressions due to Alloy limitations, in contrast to their work.

Czarnecki et al. [Czarnecki et al. 2005] introduce cardinality-based feature modeling as integration and extension of existing approaches. They specify a formal semantics for FMs and translate cardinality-based FMs into context-free grammars. Also, Antkiewicz and Czarnecki [Antkiewicz and Czarnecki 2004] present FeaturePlugin, which is a feature modeling plug-in for Eclipse. The tool supports cardinality-based feature modeling, specialization of feature diagrams, and configuration based on feature diagrams. As a future work, we intend to build a tool support to automatically translate FMs to our encoding in Alloy. In our work, we can check whether a configuration belongs to a FM. However, we do not handle cardinality-based FMs. As a future work, we intend to extend our encoding to include those kinds of constraints. In addition, their formal treatment of FM specialization could be seen as the opposite of our notion of FM refactoring.

Liu et al. [Liu et al. 2006] propose Feature Oriented Refactoring (FOR), which is the process of decomposing a (possibly legacy) program into features. Their solution does not focus on either configuration knowledge nor FMs. Also, the authors present a semi-automatic refactoring methodology to enable the decomposition of a program into features. However, FOR focuses on bootstrapping a SPL from an existing application, rather than a model, as explored in our work.

Schobbens et al. [Schobbens et al. 2007] provide a formal semantics for FMs (in their work, feature diagrams), in terms of generic formalization of syntax and semantics. This effort results in a language — Varied Feature Diagrams (VFD) — that is expressively complete, making it able to express several diverse constructs. The language bears resemblance to our encoding result, since both are FM notation-independent. Although our specification is less expressive (it lacks cardinality, for instance), we have a more specific intent, which is automatic analysis of refactorings. Several theorems from the mentioned article can be completely analyzed with our approach, such as redundancy-related theorems [Schobbens et al. 2007].

Trujillo et al. [Trujillo et al. 2006] present a case study in feature refactoring. They refactor the AHEAD Tool Suite. Feature refactoring is defined as the process of decomposing a program into a set of features. Hofner et al. [Hofner et al. 2006] propose an algebra that is used to describe and analyze the commonalities and variabilities of a system family. Sun et al. [Sun et al. 2005] propose an encoding of FMs in Alloy. It is similar to ours, but it does not consider formulas. Moreover, the previous approaches do not aim at checking meta-properties as in our work. None of those approaches aim at proposing automation of FM refactoring analysis.

Our prior work on FM encoding in Alloy [Gheyi et al. 2006b] proposes a different encoding method. Still, both encodings allow checking whether a transformation is a refactoring. However, the previous encoding [Gheyi et al. 2006b] is more detailed than this work. It specifies a single predicate of well-formedness and semantics for all FMs, different from this work in which we specify one predicate for each FM. Moreover, all first-order logic formulas are specified by signature hierarchies. As a consequence, it can check additional meta-properties, such as whether a refactoring preserves FM well-formedness rules. In contrast, the encoding presented here is much more efficient for checking FM refactorings since it contains only two signatures. Almost everything (relations, configurations, formulas) are specified using Alloy formulas instead of signatures representing all kinds of formulas. For instance, it can check whether refactorings are sound up to 10,000 features. However, since the previous encoding [Gheyi et al. 2006b] declares more signatures and recursive functions using relations, the same time period is needed to check only up to 30 features.

Segura et al. [Segura et al. 2008] use graph transformations as a suitable technology and associated formalisms to automate the merging of FMs, in which several FMs are merged into a single FM. They present a catalogue of technology-independent visual rules to describe how to merge FMs and a prototype implementation of their catalogue using the AGG system. Their FMs may include feature attributes and cross-tree constraints. In contrast, we provide FM analysis using an efficient encoding in Alloy. We consider FMs containing any kind of propositional formula, although feature attributes are not considered. In our previous work on SPL refactoring [Alves et al. 2006], we present a theory on how to merge FMs (for each initial FM, we must check whether it is refactored with respect to the final FM), and some examples of transformations. Also in that paper, we present a special kind of refactoring: *bi-directional refactoring*, in which both FMs encompass the same configurations. It is important to mention that bi-directional refactorings and merging are different kind of transformations. In this work, we focus on checking FM refactorings (a single FM is transformed into another FM), which is not tackled by Segura et al. [Segura et al. 2008]. Nevertheless, we can use our encoding for FM merging as well. In this case, the analysis would entail checking whether the target FM is a valid refactoring for each of the source FMs.

Czarnecki and Wasowski [Czarnecki and Wasowski 2007] provide an automatic and efficient procedure for generating a FM from a propositional formula. They characterize a class of logical formulas that are equivalent to FMs, and identify logical structures corresponding to the formulas syntactic elements. This procedure of generating a FM from a formula can be seen as a kind of FM refactoring that preserves configurability and improves the quality of the FM by extracting most of its formulas and expressing them graphically. With minor changes, our encoding in Alloy can be used to check whether this procedure is sound; the analysis can be carried out after specifying the initial and final FMs.

Mendonça et al. [Mendonca et al. 2008] propose a compilation technique for large scale FMs. For that, two heuristics are defined for compiling FMs to Binary Decision Diagrams (BDD). They evaluated their technique in a benchmark containing a number of FMs from the industry, literature and automatically generated ones with up to 2,000 features with 30% ECR, which represents the percentage of the features used in the propositional formulas. They performed the compilation technique on a benchmark of 10 automatically generated FM with 2,000 features and 30% ECR. As stated by the authors, the compilation process is equivalent to check whether the FM is satisfiable. Mendonça et al. evaluated their two heuristics on this benchmark. They had memory overflows in 98% and 80% of the FMs, for the respective heuristics. It took 93.9 and 58.6 seconds, respectively, to perform the analysis on the other models with non-overflowed memory. We used the same benchmark in order to evaluate the analysis performance our encoding in Alloy. We performed the satisfiability analysis on the 10 FMs and did not have memory overflows. Each analysis took less than 7 seconds. The authors also mention that they tried to evaluate their technique on FM containing 5,000 features but they had 100% of memory overflow.

Benavides et al. [Benavides et al. 2006b] also present a performance test, in this case for three off-the-shelf Java Constraint Satisfaction Problem (CSP), SATisfiability Problem (SAT) and BDD solvers. They used a benchmark containing FMs up to 300 features. They conclude that using BDDs for determining satisfiability in a FM is much faster than using SAT or CSP. We had a different result when using the benchmark presented in Mendonça et al. [Mendonca et al. 2008], with FMs containing 2,000 features. We also believe that integrating such proposals in a framework will be the right direction to follow. As a future work, we intend to use their benchmark to evaluate the performance of our encoding.

Trinidad et al. [Trinidad et al. 2008a] show how to debug inconsistent FMs. They propose a technique for automating error treatment of FMs by modeling the problems of detecting and explaining errors, and providing solutions. They consider cross-tree constraints (requires and excludes), although they do not consider any kind of propositional formula, different from our approach.

White et al. [Trinidad et al. 2008a] present how a constraint solver can derive the minimal set of feature selection changes to fix an invalid configuration. Moreover they show how this diagnosis CSP can automatically resolve conflicts between configuration participant decisions.

Our approach has a similar support for debugging FMs and configurations. In the latter case, we should specify the configuration by explicitly stating them in propositional formulas. With the Unsat Core functionality, the Alloy Analyzer 4 allows the user to find a small set of constraints that makes an Alloy model inconsistent. When performing analysis that does not yield a result when it is expected to be, the tool contains a functionality that highlights the relevant portions of the original model that contributed to the unsatisfiability. The tool guarantees that the constraints making the model inconsistent are in the portions highlighted. So, we have a limited support (we highlight a subset of the specification but we do not show explanations to avoid them) for error explanation, although the counterexamples and Unsat Core can be seen as explanations as well. But it is important to mention that this is not our goal in this work. We focus on proposing an approach for checking general FM refactorings and their application to specific FMs.

Segura et al. [Segura et al. 2010] show how to generate complex feature models representing million of products. We can use the FMs generated to evaluate the analysis performance of our approach. Finally, Benavides et al. [Benavides et al. 2010] present a literature review on the automated analysis of feature models 20 years after of their invention.

6 Conclusions

In this article, we propose an efficient encoding of feature models (FM) in Alloy. We show that this encoding, in conjunction with the Alloy Analyzer, is useful for automatically checking whether a general or specific FM refactoring is sound. Moreover, this approach not only tests whether a transformation is a refactoring, but also shows a counterexample when it is not a refactoring, which is helpful in finding likely bugs in the transformation.

We have proved a number of FM refactorings. The catalog of FM refactorings that preserve configurability is complete and minimal. It contains fourteen transformations. However the catalog that increases configurability is not complete. We need much more transformations. We can follow our previous approach and use the Prototype Verification System (PVS) [1] theorem prover to prove the new transformations [Alves et al. 2006]. If we use theorem provers, which are not simple to use, sometimes, they may need the assistance during the proofs. For example, mostly we need to provide the appropriate values when instantiating

[1] http://pvs.csl.sri.com/

the quantifications during the proofs using PVS. After that, most of them were done automatically. As a future work, we intend to further evaluate the automation by using different theorem provers. However, in this work, we prefer to use the Alloy Analyzer tool. In our approach, it was very simple and easy to use Alloy. Moreover, we checked the transformations using a scope containing thousands of features. This may be a drawback compared of using provers since we prove up to a given scope different from provers that prove for all cases. On the other hand, we check them using a scope which is reasonable in practice. Most of FMs contains less than 5,000 features. Additionally, the analysis usually takes a few seconds. Moreover, Jackson [Jackson 2006] states that the Alloy Analyzer tool usually finds a counterexample using a small scope. When it does not find a counterexample by increasing the scope, the chance that a counterexample remains does decrease. The tool performs an exhaustive search. Since we use a scope containing thousands of features, the tool performs billions of checks. This may be an indication that our transformations may be correct if we increase the scope.

Momtahan [Momtahan 2005] establishes the minimum scope an Alloy signature must have in order to yield proof in the Alloy Analyzer. Since our encoding does not present universal quantifications, we intend to check whether there is a minimum scope for *Name*. This result will be very important in order to prove FM meta-properties with *smaller scopes* in the Alloy Analyzer.

We also aim at building a FM refactoring tool, using the FAMA framework [Trinidad et al. 2008b], or extending the FeaturePlugin [Antkiewicz and Czarnecki 2004] to automatically generate Alloy specifications from a FM, and incorporate it in the FLIP tool [Soares et al. 2008], which implements some program refactorings for evolving SPLs.

Acknowledgements

We gratefully thank the guest editors, the anonymous referees, Leopoldo Teixeira, Marcílio Mendonça and Felix Chang for useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq grants 573964/2008-4 and 477336/2009-4.

References

[Alves et al. 2006] Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C. (2006). Refactoring product lines. In *Proceedings of the Generative Programming and Component Engineering*, pages 62–73, USA. ACM Press.

[Alves et al. 2007] Alves, V., Matos, Jr., P., Cole, L., Vasconcelos, A., Borba, P., and Ramalho, G. (2007). Extracting and evolving code in product lines with aspect-oriented programming. In Rashid, A. and Aksit, M., editors, *Transactions on aspect-oriented software development IV*, pages 117–142.

[Antkiewicz and Czarnecki 2004] Antkiewicz, M. and Czarnecki, K. (2004). Featureplugin: feature modeling plug-in for eclipse. In *eclipse'04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72.

[Batory 2005] Batory, D. (2005). Feature models, grammars, and propositional formulas. In *9th Software Product Lines*, volume 3714 of *LNCS*, pages 7–20. Springer.

[Benavides et al. 2005] Benavides, D., Ruiz-Cortés, A., and Trinidad, P. (2005). Automated reasoning on feature models. *17th Conference on Advanced Information Systems Engineering*, 3520:491–503.

[Benavides et al. 2006a] Benavides, D., Ruiz-Cortés, A., Trinidad, P., and Segura, S. (2006a). A survey on the automated analysis of feature models. In *XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, page 367–376, Sitges, Spain.

[Benavides et al. 2010] Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6).

[Benavides et al. 2006b] Benavides, D., Segura, S., Trinidad, P., and Ruiz-Corts, A. (2006b). A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*.

[Clements and Northrop 2001] Clements, P. and Northrop, L. (2001). *Software Product Lines : Practices and Patterns*. Addison-Wesley.

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

[Czarnecki et al. 2005] Czarnecki, K., Helsen, S., and Eisenecker, U. W. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29.

[Czarnecki and Wasowski 2007] Czarnecki, K. and Wasowski, A. (2007). Feature Diagrams and Logics: There and Back Again. In *11th International Software Product Line Conference*, pages 23–34.

[Fowler 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Garrido and Johnson 2002] Garrido, A. and Johnson, R. (2002). Challenges of refactoring c programs. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '02, pages 6–14.

[Gheyi et al. 2006a] Gheyi, R., Alves, V., Tiago Massoni, U. K., Borba, P., and Lucena, C. (2006a). Theory and proofs for feature model refactorings. Technical Report TR-UFPE-CIN-200608027, Federal University of Pernambuco. At http://www.cin.ufpe.br/spg.

[Gheyi et al. 2006b] Gheyi, R., Massoni, T., and Borba, P. (2006b). A theory for feature models in alloy. In *First Alloy Workshop*, pages 71–80, USA.

[Gheyi et al. 2008] Gheyi, R., Massoni, T., and Borba, P. (2008). Algebraic laws for feature models. *Journal of Universal Computer Science (JUCS)*, 14:3573–3591.

[Hofner et al. 2006] Hofner, P., Khedri, R., and Moller, B. (2006). Feature algebra. In *Formal Methods*, volume 4085 of *LNCS*, pages 300–315. Springer-Verlag.

[Jackson 2006] Jackson, D. (2006). *Software Abstractions: Logic, Language and Analysis*. MIT press.

[Jackson et al. 2000] Jackson, D., Schechter, I., and Shlyakhter, I. (2000). Alcoa: the alloy constraint analyzer. In *22nd International Conference on Software Engineering*, pages 730–733. ACM Press.

[Kang et al. 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute.

[Liu et al. 2006] Liu, J., Batory, D., and Lengauer, C. (2006). Feature oriented refactoring of legacy applications. In *28th International Conference on Software Engineering*, pages 112–121.

[Mendonca et al. 2008] Mendonca, M., Wasowski, A., Czarnecki, K., and Cowan, D. (2008). Efficient compilation techniques for large scale feature models. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 13–22.

[Mens and Tourwé 2004] Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

[Momtahan 2005] Momtahan, L. (2005). Towards a small model theorem for data independent systems in alloy. *ENTCS*, 128(6):37–52.

[Opdyke 1992] Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign.

[Schobbens et al. 2007] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y. (2007). Generic semantics of feature diagrams. *Computer Networks*, 51:456–479.

[Segura et al. 2007] Segura, S., Benavides, D., Ruiz-Cortés, A., and Trinidad, P. (2007). Toward automated refactoring of feature models using graph transformations. In Pimentel, E., editor, *VII Jornadas sobre Programacin y Lenguajes, PROLE2007*, pages 275–284.

[Segura et al. 2008] Segura, S., Benavides, D., Ruiz-Cortés, A., and Trinidad, P. (2008). Automated merging of feature models using graph transformations. In Lämmel, R., Visser, J., and Saraiva, J. a., editors, *Generative and Transformational Techniques in Software Engineering II*, pages 489–505.

[Segura et al. 2010] Segura, S., Hierons, R., Benavides, D., and Ruiz-Cortés, A. (2010). Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *International Conference on Software Testing, Verification and Validation*, pages 35–44.

[Soares et al. 2008] Soares, S., Calheiros, F., Nepomuceno, V., Menezes, A., Borba, P., and Alves, V. (2008). Supporting software product lines development: Flip - product line derivation tool. In *OOPSLA Companion*, pages 737–738.

[Sun et al. 2005] Sun, J., Zhang, H., and Wang, H. (2005). Formal semantics and verification for feature modeling. In *International Conference on Engineering of Complex computer systems*, pages 303–312.

[Thum et al. 2009] Thum, T., Batory, D., and Kastner, C. (2009). Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, pages 254–264.

[Torlak et al. 2008] Torlak, E., Chang, F. S.-H., and Jackson, D. (2008). Finding minimal unsatisfiable cores of declarative specifications. In *Proceedings of the 15th international symposium on Formal Methods*, FM '08, pages 326–341.

[Trinidad et al. 2008a] Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., and Toro, M. (2008a). Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81:883–896.

[Trinidad et al. 2008b] Trinidad, P., Benavides, D., Ruiz-Cortés, A., Segura, S., and A.Jimenez (2008b). Fama framework. In *12th Software Product Lines Conference (SPLC).*

[Trujillo et al. 2006] Trujillo, S., Batory, D., and Diaz, O. (2006). Feature refactoring a multi-representation program into a product line. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 191–200.

[van Deursen and Klint 2002] van Deursen, A. and Klint, P. (2002). Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17.

[White et al. 2008] White, J., Schmidt, D. C., Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2008). Automated diagnosis of product-line configuration errors in feature models. In *Proceedings of the 2008 12th International Software Product Line Conference*, pages 225–234.