A Theory for Feature Models in Alloy

Rohit Gheyi
rg@cin.ufpe.br

Tiago Massoni
tlm@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Brazil

ABSTRACT

Feature models are used to state the instances of a software product-line. However, there is a limited tool support for automatically checking properties of feature models. In this paper, we propose a theory of feature models in Alloy. This theory can be used to check a number of properties in the Alloy Analyzer. For instance, we show how to check whether general feature model transformations preserve the well-formedness rules of feature models. This theory is compared with an alternative theory in Alloy for checking feature model refactorings.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model Checking; D.2.13 [**Reusable Software**]: Domain engineering

General Terms

Design, Languages, Verification

Keywords

Model Checking, Software Product Lines, Feature Model

1. INTRODUCTION

A software product line (SPL) [5] is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission, being developed from a common set of core assets in a prescribed way.

Feature modeling is a technique for analyzing and capturing the common and variable features of systems in a system family and their interdependencies. The results of feature modeling are captured in a feature model (FM) [6]. In Section 2, we give an overview of FMs. However, developers have a limited tool support for automatically checking properties of feature models. So, developers mostly do manual reasoning, which is usually difficult, time-consuming and error-prone.

In this paper, we propose a theory of FMs in Alloy, as explained in Section 3. We show how this theory can be used to check a number of properties in the Alloy Analyzer up to a given scope, as shown in Section 4. For instance, we check whether general feature model transformations preserve the well-formedness rules of feature models. Moreover, we show that it is possible to check general properties of the FM refactoring relation, such as reflexivity and transitivity. The main contributions of this paper are the following:

- A theory for feature models in Alloy (Section 3);
- We evaluate this theory showing a number of feature model properties that can be analyzed in the Alloy Analyzer (Section 4).

We compare this general theory (G-Theory) with another one (R-Theory), which is useful for checking feature model refactorings, in Section 5. G-Theory is useful for checking general properties of a FM. In contrast, R-Theory was specified for checking FM refactorings. R-Theory cannot check some properties that can be checked in G-Theory. For example, G-Theory can check whether general feature model transformations preserve the well-formedness rules of feature models. This property cannot be checked in R-Theory. G-Theory can also be useful for checking FM refactorings. However, since R-Theory is specific for checking FM refactorings, it can perform much faster analysis and use a greater scope than G-Theory.

2. FEATURE MODELS

In this section, we give an overview of FMs. A FM represents the common and variable features of a SPL and the dependencies between them [6]. A feature diagram, which is a tree, is a graphical representation of a feature model.

Relationships between a parent feature and its child features (or subfeatures) are categorized as: *Optional* (features that are optional), *Mandatory* (features that are required), *Or* (one or more must be selected - represented by a filled triangle), and *Alternative* (exaclty one subfeature must be selected - represented by a unfilled triangle). Figure 1 depicts these relationships graphically.

Besides these relationships, we allow FMs to include propositional formulas about features. For instance, the formula $earphone \Leftrightarrow mp3$ states that the feature $earphone$ is selected iff the feature $mp3$ is selected.

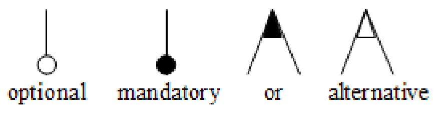


Figure 1: Feature Diagram Notations

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

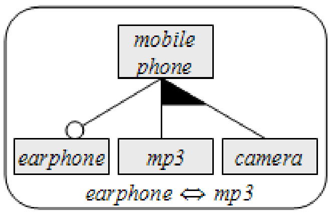


Figure 2: Feature Diagram Example

Semantics. The semantics of a FM is the set of its possible (valid) configurations. A 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. However, the configuration $\{$ *mobilephone*, *earphone* $\}$ is invalid because the or feature relation between *mobilephone*, *mp3* and *camera* states that whenever *mobilephone* is selected, *mp3* or *camera* must be selected.

Refactorings. The traditional definition of program refactoring [15, 8] does not take into account intrinsic characteristics of SPL: feature models and configuration knowledge mapping instances of the feature model (FM) to classes and aspects in the solution space [6]. For instance, refactoring of a SPL may have the undesirable effect of reducing its configurability. Another problem is that the traditional notion of refactoring applies only to a single product rather than to a SPL, thereby not taking into account transformations involving more than one product. Therefore, the standard definition of refactoring needs to be extended for SPLs, taking into account their specific characteristics.

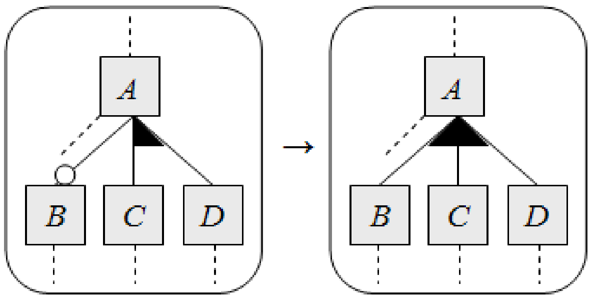
We extended the traditional notion of refactoring to the SPL context [1]. Besides traditional program refactoring, feature models are refactored. Not only the quality of the program is improved, but also the quality of the feature model. We define a FM refactoring as *a transformation that improves the quality of a FM by improving (maintaining or increasing) its configurability* [1]. So the resulting FM contains all valid configurations of the initial FM.

A number of FM refactorings are proposed that can be applied based on template matching [1]. Next we give an overview of the notation used to state them.

Notation: Each refactoring consists of two templates (patterns) of FMs, on the left-hand (LHS) and right-hand (RHS) sides. We can apply a refactoring whenever the left template is matched by the FM. A matching is an assignment of all meta-variables occurring in LHS/RHS models to concrete values. Any element not mentioned in both FMs remains unchanged, so the refactoring templates only show the differences between the 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.

For instance, Refactoring 2 collapses an optional feature and an or feature relation into a general or feature relation encompassing all features. It is important to mention that A, B, C and D are meta-variables.

Refactoring 2. ⟨*collapse optional and or*⟩



For instance, the FM depicted in Figure 2 matches the LHS FM template of Refactoring 2. The meta-variables A, B, C and D are matched with *mobilephone*, *earphone*, *mp3* and *camera*, respectively. Therefore, we can apply this refactoring to the FM depicted in Figure 2 in order to collapse an optional feature and an or feature relation into a general or feature relation encompassing all features.

3. G-THEORY

Next we specify an abstract syntax (Section 3.1), well-formedness rules (Section 3.2) and semantics (Section 3.3) for FMs in Alloy. In Section 3.4, we show how to specify a FM in this *general theory* (G-Theory). Our aim is to perform analysis in FMs using the Alloy Analyzer.

3.1 Abstract Syntax

A FM has a set of features and a root node feature. A root should be one of the features of the FM. A feature is represented by its name. Moreover, a FM may declare a set of relations and formulas.

```
sig FM {
  features: set Name,
  root: features,
  relations: set Relation,
  forms: set Formula
}
sig Name {}
```

A FM may include four types of relations (Figure 1): optional, mandatory, or and alternative. We represent them by the following signatures.

```
sig Relation {
  parent: Name,
  child: set Name,
  type: Type
}
abstract sig Type {}
one sig Optional, Mandatory, OrFeature,
        Alternative extends Type {}
```

Finally, a FM may declare formulas. We consider FMs declaring propositional logic formulas. In Sections 3.2 and 3.3, we check whether a formula is well-typed in a FM, and a configuration satisfies a formula, using recursive functions. However, Alloy 3 does not support recursive functions. In order to solve this problem, we declare two relations (`satisfy` and `wt`) representing recursive functions. In Sections 3.2 and 3.3, we explain their constraints in more detail. Notice that we are using the `Bool` signature, which represents a boolean.

```
abstract sig Formula {
  satisfy: Configuration -> one Bool,
  wt: FM -> one Bool
}
```

Next, we specify propositional formulas. A feature name is a formula itself represented by the `NameF` signature. Moreover, the `Form` signature declares a general binary formula.

```
sig NameF extends Formula {
  n: Name
}
sig Form extends Formula {
  f: Formula,
  g: Formula,
  op: Op
}
abstract sig Op {}
one sig AndF, OrF, ImpliesF, NotF extends Op {}
```

In fact, we could declare one signature for each kind of formula. However, we prefer to declare the `Op` signature representing formulas. We also declare unary formulas, such as not, in `Form`. In this case, we assume an unconstrained `g` relation.

3.2 Well-Formedness Rules

In this section, we specify some well-formedness rules for FMs. A FM is well-formed iff all relations are well-formed, and formulas are well-typed, as specified next. Notice that we check whether formulas are well-typed using the ternary relation (`wt`) declared in Section 3.1, which represents a recursive function.

```
pred wellFormed(fm:FM) {
  wellFormedRel(fm)
  all f:fm.forms | f.wt[fm] = True
}
```

All features in a relation must be declared by the FM (line 1). The optional and mandatory relations relate a parent feature to exactly one subfeature (line 2), whereas the or and alternative relations relate a parent feature to at least two subfeatures (line 3), as specified by the following predicate.

```
  pred wellFormedRel(fm:FM) {
    all r:fm.relations {
1     r.parent+r.child in fm.features
2     r.type in Optional+Mandatory => #r.child=1
3     r.type in Alternative+OrFeature => #r.child>1
    }
  }
```

We have other constraints stating that all relations of a FM represent a tree, not shown here for simplicity. In most situations, we are going to perform analysis on specific FMs, which are assumed to satisfy this constraint.

Next, we specify when formulas are well-typed. First of all, we do not allow an atom of `Form` to refer to itself, avoiding circularity. We do not want the same atom being related to itself by the `f` and `g` relations. It is important to mention that we can have two different atoms of `NameF` being related to the same name by the `n` relation. A `NameF` formula is well-typed in a FM iff its name is declared by the FM.

```
fact FormulaConstruction {
  all form:Form | form !in form.^(f+g)

  all fm:FM {
    all f:NameF | f.wt[fm] = satisfyNameWT(f,fm)
    all f:Form | f.wt[fm] = satisfyFormWT(f,fm)
  }
}
fun satisfyNameWT(form:NameF, fm:FM): Bool {
  if (form.n in fm.features) then True
  else False
}
```

In order to check whether a `Form` formula is well-typed in a FM, we check the unary negation and binary formulas. A negation formula is well-typed iff the `f` relation is well-typed. Binary formulas are well-typed iff the `f` and `g` relations are well-typed, as specified next. Notice that we use the `And` predicate declared in the `Bool` specification.

```
fun satisfyFormWT(form:Form, fm:FM): Bool {
  if(form.op=NotF) then
    form.f.wt[fm]
  else And(form.f.wt[fm],form.g.wt[fm])
}
```

3.3 Semantics

In this section, we specify semantics for FMs. A configuration, which contains a set of feature names (selected for a given software product), is represented by the following signature. It is important to notice that this signature is a *datatype* [14]. A datatype has an implicit *generator axiom*, which generates all possible combinations, as stated by the fact `configDatatype`.

```
sig Configuration {
  value: set Name
}
fact configDatatype {
  all n:set Name | some c:Configuration | c.value=n
}
```

The semantics of a FM is the set of all possible configurations that satisfy all relations, implicit and explicit constraints, as specified next.

```
fun semantics(fm:FM): set Configuration {
  {c:Configuration |
     satisfyRelations(fm,c) and
     satImpConst(fm,c) and
     satExpConst(fm,c)
  }
}
```

Satisfying Relations. Next we show when a configuration satisfies all relations of a FM. For instance, in a mandatory relation, if the parent feature is selected in the configuration, then the child subfeature must also be selected, and vice versa. As another example, in an alternative relation, if the parent feature is selected, then exactly one child subfeature must be selected in the configuration. The other relations (optional and or) are specified similarly.

```
pred satisfyRelations(fm:FM, c:Configuration) {
  all r:fm.relations {
    (r.type=Optional =>
      (r.child in c.value => r.parent in c.value))
    (r.type=Mandatory =>
      (r.child in c.value <=> r.parent in c.value))
    (r.type=Alternative =>
      (r.parent in c.value =>
        one n:r.child | n in c.value))
    (r.type=OrFeature =>
      (r.parent in c.value =>
        some n:r.child | n in c.value))
  }
}
```

Implicit Constraints. A FM has two implicit constraints. A FM root must be selected in all configurations. Moreover, the features selected in a configuration must be declared by the FM.

```
pred satImpConst(fm:FM, c:Configuration) {
  fm.root in c.value
  c.value in fm.features
}
```

Explicit Constraints. Finally, a configuration satisfies explicit constraints of a FM iff it satisfies all its formulas.

```
pred satExpConst(fm:FM, c:Configuration) {
  all f:fm.forms | f.satisfy[c] = True
}
```

As mentioned in Section 3.1, the ternary relation `satisfy` represents a recursive function. It checks whether a formula satisfies a configuration. For example, a configuration satisfies the `NameF` formula iff the configuration contains its name.

```
fact FormulaSatisfaction {
  all c: Configuration {
    all f: NameF | f.satisfy[c] = satisfyName(f,c)
    all f: Form | f.satisfy[c] = satisfyForm(f,c)
  }
}
fun satisfyName(f:NameF, c:Configuration): Bool {
  if(f.n in c.value) then True
  else False
}
```

The binary formulas are checked similarly. For example, a configuration satisfies an and formula iff it satisfies both relations (`f` and `g`). As another example, a configuration satisfies a negation formula iff it does not satisfy the `f` relation. We used some predicates declared in the `Bool` specification. The satisfaction of or and implies formulas are specified similarly.

```
fun satisfyForm(f:Form, c:Configuration): Bool {
  if (f.op = AndF) then satisfyAnd(f,c) else
  if (f.op = NotF) then satisfyNot(f,c) else
  if (f.op = OrF) then satisfyOr(f,c)
  else satisfyImplies(f,c)
}
fun satisfyAnd(form:Form, c:Configuration): Bool {
  And(form.f.satisfy[c],form.g.satisfy[c])
}
fun satisfyNot(form:Form, c:Configuration): Bool {
  Not(form.f.satisfy[c])
}
```

3.4 Example

In this section, we show how we can specify the FM from Figure 2 using the theory presented before. Firstly, we declare its elements. We declare a singleton (`one`) subsignature for the FM of Figure 2, which is represented by the subsignature `fm1`. Moreover, we declare a singleton subsignature for each feature name. Finally, the FM in Figure 2 depicts two relations (optional and or feature) and two formulas (the biimplication will be represented by two implications). We declare a singleton subsignature for each of them.

```
one sig mobilephone,earphone,mp3,camera extends Name {}
one sig fm1 extends FM {}
one sig r1,r2 extends Relation {}
one sig f1,f4 extends Form {}
```

In the following fact we specify all elements of `fm1`. Its root is `mobilephone`. Moreover, it declares four features: `mobilephone`, `earphone`, `mp3` and `camera`. Finally, it declares two relations and two formulas, which are specified next.

```
fact elements {
  fm1.root = mobilephone
  fm1.features = mobilephone+earphone+mp3+camera
  fm1.relations = r1+r2
  fm1.forms = f1+f4
}
```

Figure 2 depicts two relations: optional and or. The optional relation between `mobilephone` and `earphone` is represented by the `r1` relation, whereas the `r2` relation represents the or relation between `mobilephone`, and `mp3` and `camera`.

```
fact relations {
  r1.type = Optional
  r1.parent = mobilephone
  r1.child = earphone

  r2.type = OrFeature
  r2.parent = mobilephone
  r2.child = mp3+camera
}
```

Finally, we specify the explicit formula using our theory. We must declare a formula for each subformula of *earphone* ⇔ *mp*3. So, we must declare a `NameF` formula for each name in the formula. In our example, we must create two formulas for `earphone` and `mp3`, which are represented next by the formulas `f2` and `f3`. Moreover, we must create a formula for each operator. In the *earphone* ⇔ *mp*3 formula, the biimplication is represented by two implications. Therefore, we must specify two formulas (`f1` and `f4`).

```
one sig f2,f3 extends NameF {}

fact formulas {
  f1.op = ImpliesF
  f1.f = f2
  f1.g = f3

  f2.n = earphone
  f3.n = mp3

  f4.op = ImpliesF
  f4.f = f3
  f4.g = f2
}
```

Observe that this approach is systematic and can be implemented by a tool. So far we have shown a theory for FMs and how we can use it to specify FMs in Alloy. Next we are going to explain how it can be used to perform analyses on FMs using the Alloy Analyzer tool.

4. ANALYSES

In this section we show analyses that can be performed in the previous theory.

4.1 Configurations

It is useful to perform analysis on specific models, such as the FM from Figure 2. Figure 2 presents exactly one FM, four feature names, two relations and four formulas. Since we know exactly the number of elements of each signature declared in Section 3.1, we can perform a *complete analysis* using the Alloy Analyzer. Next, we show two analyses that can be performed on the example presented in Section 3.4.

1st Analysis (Checking a Configuration Against a FM). It may be useful to check whether a specific configuration is valid for a FM. For example, the following assertion checks whether selecting all features but `mp3`, which is represented by the `Config1` configuration, is a valid configuration for the FM of Figure 2.

```
one sig Config1 extends Configuration {}{
  value=mobilephone+earphone+camera
}
```

```
assert validConfig {
  Config1 in semantics(fm1)
}
check validConfig for 1 FM, 4 Name, 4 Formula,
                      2 Relation, 16 Configuration
```

The number of elements of `Configuration` is 2^n, where n is the number of elements of `Name`. As explained before, `Configuration` is a datatype. Since the FM of Figure 2 has four names, we must perform analysis using 16 configurations.

The analysis on the `validConfig` assertion shows a counterexample indicating that this assertion is not valid. So selecting `mobilephone`, `earphone` and `camera` features is not a valid configuration for `fm1`. As another example, running the previous assertion selecting all features does not yield a counterexample. Therefore, it is a valid configuration of `fm1`. Since the Alloy Analyzer performs a *complete analysis*, this result is a theorem.

2nd Analysis (Finding all Configurations). Besides checking whether a configuration is valid for a FM, the Alloy Analyzer can be used to find all valid configurations of a FM. Suppose that one would like to know all valid configurations of the FM depicted in Figure 2. Running the `show` function, which is declared next, yields all configurations of `fm1`. We use the same scope of `validConfig`. The `fm1` FM has three valid configurations, as depicted in Figure 3.

```
fun show():set Configuration {
  semantics(fm1)
}
```

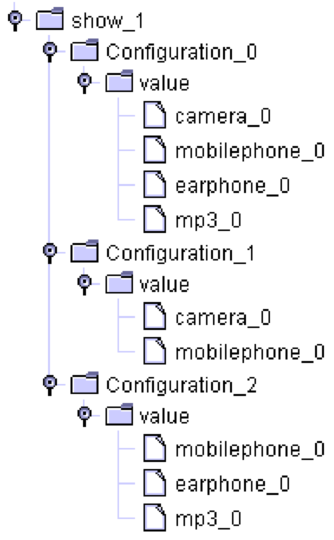


Figure 3: Valid Configurations

We used a laptop with 2GHz processor and 1GB RAM memory to perform analysis. Both analyses take less than 20 seconds to be performed. Since analysis on specific FMs are the most common, we will exactly know the number of elements of all signatures declared in Section 3.1. Therefore, we can perform a complete analysis.

4.2 Properties

As explained in Section 2, a FM refactoring improves the quality of a feature model by improving (maintaining or increasing) its configurability. So, according to definition, `fm2` refactors `fm1` iff all valid configurations of `fm1` are valid configurations of `fm2`, as formalized by the following predicate.

```
pred refactoring(fm1,fm2:FM) {
  all c:Configuration |
    c in semantics(fm1) => c in semantics(fm2)
}
```

Based on the previous definition, we can check general properties of this predicate. For instance, the `Reflexive` and `Transitive` assertions check whether the `refactoring` predicate is reflexive and associate for a predefined scope, respectively.

```
assert Reflexive {
  all fm:FM | refactoring(fm,fm)
}
check Reflexive for 5 but 32 Configuration
assert Transitive {
  all fm1,fm2,fm3:FM |
    refactoring(fm1,fm2) and refactoring(fm2,fm3) =>
      refactoring(fm1,fm3)
}
check Transitive for 5 but 32 Configuration
```

Each analysis takes less than a minute to be performed. Notice that we are checking meta-properties (general properties) of the FM theory. The `refactoring` predicate is reflexive and transitive for any two FMs containing less than six features.

We can automatically prove that `refactoring` is reflexive and transitive using a good theorem prover, such as the PVS prover [16]. However, we may check in the Alloy Analyzer other non-trivial general properties. For example, we may check whether a general FM transformation does not break well-formedness rules of FMs. In this situation, the counterexamples generated by the Alloy Analyzer are very important in order to improve our understanding. After gaining some confidence that the property is valid, we may use a theorem prover to prove it.

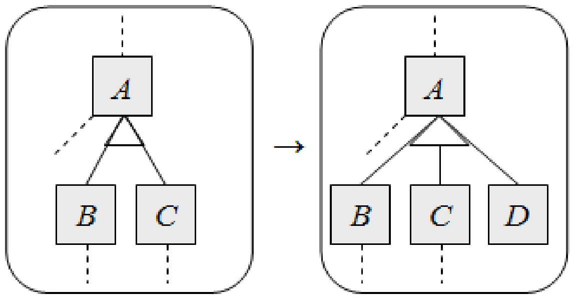
4.3 FM Refactorings

Another work proposes a catalog of FM refactorings [1]. Those refactorings are proved sound with respect to a formal semantics in a theorem prover [12]. A technical report [12], which has more than 700 pages of proofs, proves 12 FM refactorings.

As is widely known, it is difficult to use a theorem prover. Moreover, proving refactorings manually may be time-consuming and error-prone. Next we show how we can automatically check whether refactorings are correct up to a given scope.

For instance, Refactoring 5 allows us to add a new node D and increase the alternative between B, C and D [1]. This refactoring can be applied to any FMs that match the templates. Next we show how to specify Refactoring 5 in our theory presented in Section 3.

Refactoring 5. ⟨*add new alternative*⟩



4.3.1 *Specification*

The LHS and RHS FMs are represented by `fm1` and `fm2`, respectively. Moreover, Refactoring 5 includes four feature names: `A`, `B`, `C` and `D`. Finally, two relations are present: an alternative between `A`, `B` and `C` in the LHS FM, and an alternative between `A`, `B`, `C` and `D` in the RHS FM. Notice that Refactoring 5 does not have any explicit formula.

```
one sig fm1,fm2 extends FM {}
one sig A,B,C,D extends Name {}
one sig r1,r2 extends Relation {}
```

Next we specify the syntactic relationship of both FMs in Refactoring 5. As explained in Section 2, any element not mentioned in both models is equivalent. Both FMs have the same *root* and *formulas*. Moreover, since `B`, `C` and `D` are subfeatures, we must state that they cannot be a root of a FM.

```
fact SyntacticRelationship {
  fm1.root = fm2.root
  fm1.forms = fm2.forms
  fm1.root !in B+C+D
```

Moreover, both models have the same *features*, except for the `D` feature, which does not belong to the LHS FM. Both models have the `A`, `B` and `C` features, as formalized next.

```
  fm1.features = fm2.features - D
  (A+B+C) in fm1.features
  (A+B+C+D) in fm2.features
```

Finally, we specify all *relations* in both FMs. They have the same relations expect for two alternatives, which are represented by `r1` and `r2`.

```
  fm1.relations - r1 = fm2.relations - r2
  r1 in fm1.relations
  r2 in fm2.relations

  r1.type = Alternative
  r1.parent = A
  r1.child = B+C

  r2.type = Alternative
  r2.parent = A
  r2.child = B+C+D
}
```

After stating both FMs in our theory, we can write an assertion and check whether Refactoring 5 is sound up to a given scope. For instance, the following assertion states that if the LHS FM is well-formed, then the RHS FM refactors the LHS FM. In order to apply a refactoring, we assume that the initial FM should always be well-formed. We do not want to refactor an ill-formed FM.

```
assert refactoringLR {
  wellFormed(fm1) => refactoring(fm1,fm2)
}
check refactoringLR for 5 but 32 Configuration
```

When proposing refactorings, *refactoring designers*, which are responsible for proposing refactorings, should also prove that all FMs resulted by applying a refactoring must also be well-formed. In practice, developers do not want to apply a refactoring, which results an ill-formed FM. Next we specify an assertion in order to check this property.

```
assert wfPreservation {
  wellFormed(fm1) => wellFormed(fm2)
}
check wfPreservation for 5 but 32 Configuration
```

We can similarly check whether any kind of FM transformation, not only refactorings, preserves the well-formedness rules.

4.3.2 Analysis

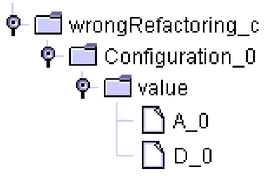
The `refactoringLR` and `wfPreservation` assertions do not yield a counterexample for Refactoring 5. So, we can apply Refactoring 5 for any FMs containing less than 6 features, and we have a guarantee that the refactoring is sound and all FMs resulted are well-formed. Each analysis takes less than a minute to be performed. We have specified in our theory all refactorings proposed elsewhere [1].

Sometimes refactoring designers may propose transformations that are intended to be a refactoring, but they are not. In those situations, the Alloy Analyzer shows a counterexample. For instance, suppose that the refactoring designer would like to know whether applying Refactoring 5 from right to left also defines a refactoring.

```
assert wrongRefactoring {
  refactoring(fm2,fm1)
}
check wrongRefactoring for 5 but 32 Configuration
```

Checking the previous assertion yields a counterexample depicted in Figure 4. We can select features $\{A, D\}$ in the RHS FM but we cannot select them in the LHS FM, because the feature `D` does not belong to it. Since the resulting FM does not contain all valid configurations of the initial FM, it is not a refactoring. This counterexample helps refactoring designers to improve the understanding why the transformation is not a refactoring.

Undesirably, the scope of `Configuration` exponentially increases with the number of feature names. In order to



```
wrongRefactoring_c
  Configuration_0
    value
      A_0
      D_0
```

Figure 4: Counterexample

solve this problem, we remove the `configDatatype` fact and change the `semantics` predicate by checking whether a specific configuration satisfies the semantics of a FM, instead of generating all valid configurations. In this way, we can check whether a FM transformation is a refactoring using the following assertion. This is an optimization in order to increase the scope of the analysis. For example, in this way we can check whether Refactoring 5 is sound using a scope of 20 features, relations and formulas.

```
pred semantics2(fm:FM, c:Configuration) {
  satImpConst(fm,c)
  satExpConst(fm,c)
  satisfyRelations(fm,c)
}
assert refactoringOptimized {
  all c:Configuration |
     wellFormed(fm1) and semantics2(fm1,c) =>
       semantics2(fm2,c)
}
check refactoringOptimized for 20
```

4.3.3 Generalization

So, for specifying and checking other FM refactorings, refactoring designers just have to specify the following *hot spots* for each refactoring:

- extend `Name` with all features' names in the refactoring;
- extend `Relation` with all relations in the refactoring;
- extend `Formula` with all formulas in the refactoring;
- specify a syntactic relationship (`root`, `features`, `relations` and `forms`) between both FMs.

This approach is systematic. We can build a tool to automatically generate a specification in our theory from a FM refactoring.

5. DISCUSSION

In Section 5.2, we compare the theory proposed for FMs in Section 3 and another one [11], which is briefly explained next.

5.1 R-Theory

We have proposed another theory for FMs in Alloy [11] useful for checking FM refactorings (R-Theory). In this theory, a FM has a set of feature names. In our theory, we only have two signatures (`FM` and `Name`) representing all elements of a FM.

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

Relations and formulas are represented by predicates. For instance, suppose that the parent feature `A` is related to a subfeatures `B` and `C`. A configuration (`config`) is represented by a set of names. The following predicates represent the mandatory and or feature relations. The other relations are declared similarly.

```
pred mandatory(A,B:Name, config:set Name) {
  A in config <=> B in config
}
pred orFeature(A,B,C:Name, config:set Name) {
  A in config <=> (B in config) or (C in config)
}
```

This G-Theory abstracts from the syntax of formulas, which are directly encoded based on semantics. Each formula's operator is directly translated to equivalent one in Alloy. For example, the $B \Rightarrow \neg C$ formula is expressed by the following Alloy fragment. Every occurrence of a feature name is appended with `in config`.

```
  B in config => !(C in config)
```

Finally, in this theory, we should specify a semantics and refactoring relations for each FM. Moreover, this theory is not concerned with well-formedness rules.

5.2 Comparison

In this section, we compare the theories presented in Sections 3 and 5.1.

Abstract Synstax. G-Theory (Section 3) represents all elements of a FM in signatures. For instance, it declares the `Formula` and `Relation` signatures representing the abstract syntax of a number of propositional formulas, and the FM relations, respectively. R-Theory represents all FM relations and formulas using Alloy predicates and formulas.

Well-formedness Rules. G-Theory specifies when a formula is well-typed (the `wt` relation). R-Theory abstracts the well-formedness rules of FMs.

Semantics. G-Theory declares the `semantics` predicate for any FM. In R-Theory, we need to specify a semantics predicate for each FM. However, this approach is systematic and can be easily automated by a tool [11].

Analyses. Both theories allow us to check whether a configuration belongs to a FM. Moreover, G-Theory can yield all valid configurations for a specific FM. This is not possible in R-Theory. In the current version of the Alloy Analyzer, we cannot perform analysis using more than 5 feature names since the tool crashes. The number of elements of `Configuration` exponentially increases.

G-Theory specifies general predicates (`semantics` and `refactoring`) representing semantics and refactoring relations, respectively. We can use the Alloy Analyzer to check general properties, such as reflexivity and transitivity, about the refactoring relation. Since R-Theory needs to specify a specific semantics predicate for each FM, we cannot check general properties.

In both abstractions we can check whether a transformation is a refactoring. Since G-Theory specifies when a formula is well-typed, we can also check whether a transformation preserves the well-formedness rules of a FM. This property cannot be checked in R-Theory.

However, since R-Theory has only two signatures (`FM` and `Name`), we can use a greater scope than G-Theory to check whether a transformation is a refactoring. Notice that R-Theory represents a configuration as a set of names. So we do not declare a configuration signature. Therefore, we avoid the `configDatatype` fact, which is responsible for exponentially increasing the `Configuration` scope.

Using an optimized version (Section 4.3.2) of G-Theory, we can perform analysis using a scope with 20 feature names, relations and formulas. For instance, the Alloy Analyzer takes 20 minutes to check that Refactoring 5 is correct, and it preserves the well-formedness rules.

We also used R-Theory to check whether refactorings are correct. As mentioned before, we could not check whether they preserve the well-formedness rules. We used a laptop with 2GHz processor and 1GB RAM memory to perform analysis. Figure 5 presents the analysis performance of eleven refactorings proposed [1] up to 300 features. Since we represent relations and formulas in predicates, we can have any number of relations and formulas. The most important information in Figure 5 is to know the maximum amount of time required to check a refactoring. It takes 8 minutes (maximum value) to check Refactoring 5, which adds a new alternative, using 300 features. Refactoring 12, which adds an optional node, is checked in 6 minutes (minimum value). It is important to mention that refactoring designers just need to check them once. After that developers can use FM refactorings whenever they are dealing with FMs containing less than or equal to 300 features. Moreover, since in practice most of FMs have less than 300 features, we can apply them most of the time. Some of the FM refactorings proposed [1] were checked using a scope of 400 features. Each analysis takes between 10 and 15 minutes.

All refactorings [12] presented in Figure 5 were proved using the PVS prover [16] containing more than 700 pages of proofs. In this theory, we can automatically check all eleven refactorings (up to 300 features) in at most two hours using the Alloy Analyzer.

G-Theory is very useful for checking general properties of FMs. We can extend it with other relations and check properties. In contrast, R-Theory is very useful for checking FM refactorings. It is much faster and can use a greater scope than G-Theory. However, R-Theory cannot check general properties of FMs. So, we should select the appropriate theory depending on the property that we would like to check.

6. RELATED WORK

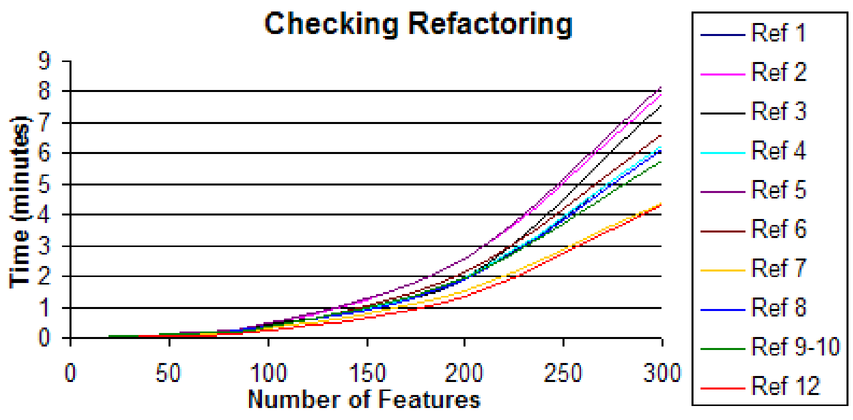


Figure 5: Performance of Checking FM Refactorings

Batory [3] integrates prior results to connect feature diagrams, grammars, and propositional formulas. This connection also allows the use SAT solvers to help debug feature models by confirming compatible and incomplete feature sets. He explains in more details how to check three properties: (1) a FM has a contradiction, (2) the user selects some features and the tool yields a value to the remaining features that satisfies the FM semantics, and (3) a configuration is a valid for a FM. All three properties can be checked in the Alloy Analyzer. We explained how to check the first and third property in Section 4.1. If the semantics relation does not yield any configuration, we have a contradiction (inconsistency). The second property can be checked using the following predicate. The Alloy Analyzer yields all possible valid configurations of a FM that selects `mobilephone` and `camera`.

```
fun show2():set Configuration {
  { c:Configuration |
      c in semantics(fm1) and
      (mobilephone+camera) in c.value
  }
}
```

Besides these properties, we show how the Alloy Analyzer can be useful for checking meta-properties. Batory does not define a general FM semantics definition. He follows a similar approach of R-Theory. Since he does not have a general definition of semantics, he could not check meta-properties of this relation that can be checked in G-Theory. In G-Theory, we can specify other general relations, such as refactoring, and check meta-properties about them using the Alloy Analyzer.

Czarnecki et al. [7] introduces cardinality-based feature modeling as an integration and extension of existing approaches. They specify a formal semantics for FMs with these features and translate cardinality-based FMs into context-free grammars. Another work [2] presents a Feature-Plugin, which is a feature modeling plug-in for Eclipse. This plug-in implements cardinality-based feature modeling [7]. The tool supports cardinality-based feature modeling, specialization of feature diagrams, and configuration based on feature diagrams. In our work, we can check whether a configuration belongs to a FM. G-Theory and R-Theory do not handle cardinality-based FMs. Our work can check meta-properties of FMs that cannot be check in their work, which focuses on checking properties of specific FMs. We also note that their formal treatment of FM specialization could be seen as the opposite of our notion of FM refactoring. So, we believe that specializations can be analyzed similarly. Another work [13] proposes an algebra that is used to describe and analyze the commonalities and variabilities of a system family.

A related approach [4] proposes an automatic way to analyze five properties of FMs, such as yield the number of instances and all instances of a FM, and check whether a FM is valid. They present a mapping to transform an extended feature model into a Constraint Satisfaction Problem in order to formalize extended feature models using constraint programming. In our work, we can check these properties. Their idea of filters is equivalent to formulas in our FMs. Moreover, we can also check using Alloy Analyzer some meta-properties of FMs that they cannot check. Our theory has a limited support for integer expressions due to Alloy, in contrast to their work.

Another work [17] proposes a textual language for describing features. Their language is similar to ours, but do not consider formulas. They propose a notion of FM semantics that is equivalent to ours. Also, a number of rules relating equivalent FMs are proposed, which are very similar to bidirectional refactorings. They informally argue soundness, in contrast with our approach, which uses the Alloy Analyzer to increase the confidence.

7. CONCLUSIONS

In this paper, we propose a theory for FMs in Alloy. This theory is useful for automatically checking a number of properties in the Alloy Analyzer. For instance, we show how to yield all valid configurations of a FM. Moreover, we explain how to check whether FM transformations preserve well-formedness rules of a FM. Additionally, we used the Alloy Analyzer to check general properties. For instance, we show that the refactoring relation is reflexive and symmetric using a scope of 5 atoms. Finally, we compared G-Theory and R-Theory [11], which is useful for checking FM refactorings. We cannot check some properties in R-Theory that can be checked in G-Theory. However, when checking FM refactorings, R-Theory is much faster and can use a greater scope than G-Theory.

We can use a greater scope and perform faster analyses with the evolution of the Alloy Analyzer and SAT Solvers. Comparing the analysis performance of the current version of the Alloy Analyzer to the last year's version, we notice at least a 20% of improvement.

As a future work, we aim at building a tool allowing us to perform analysis on FMs. The user will select one of the FM theories proposed, and the tool will automatically generate an Alloy specification from a FM. Moreover, we will investigate whether directly translating G-Theory to SAT may allow us to increase the scope instead of using the Alloy Analyzer's translation.

Another work has proposed a set of sound object model refactorings for Alloy [9, 10]. We also intend to check whether some models specified in G-Theory can be refactored to R-Theory using the object model refactorings proposed. Since R-Theory can perform much faster analysis and use a greater scope, we will be able to leverage those benefits to G-Theory.

Acknowlegments

We would like to thank all anonymous referees and members of the Software Productivity Group at UFPE, for their important comments. This work was supported by CNPq and CAPES (Brazilian research agencies).

8. REFERENCES

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *Proceedings of the Generative Programming and Component Engineering*, United States, 2006.

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

[3] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference of Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

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

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

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

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

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

[9] R. Gheyi, T. Massoni, and P. Borba. Basic laws of object modeling. In *3rd Specification and Verification of Component-Based Systems*, pages 18–25, United States, 2004.

[10] R. Gheyi, T. Massoni, and P. Borba. A rigorous approach for proving model refactorings. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 372–375, United States, 2005.

[11] R. Gheyi, T. Massoni, and P. Borba. Automatically checking the correctness of feature model refactorings. Submitted for Application, 2006.

[12] R. Gheyi, T. Massoni, and P. Borba. Theory and proofs for feature model refactorings in PVS. Technical report, Federal University of Pernambuco, 2006.

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

[14] V. Kuncak and D. Jackson. Relational analysis of algebraic datatypes. In *Proceedings of the 13th Foundations of software engineering*, pages 207–216. ACM Press, 2005.

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

[16] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. PVS: an experience report. In *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Germany, 1998. Springer-Verlag.

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