Journal of Universal Computer Science, vol. 14, no. 21 (2008), 3573-3591
submitted: 16/4/08, accepted: 5/6/08, appeared: 1/12/08 © J.UCS

Algebraic Laws for Feature Models

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

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

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

Abstract: Software Product Lines (SPL) may be adopted by either bootstrapping existing software products into a SPL, or extending an existing SPL to encompass an additional software product. Program refactorings are usually applied for carrying out those tasks. The notion of SPL refactoring is an extension of the traditional definition of refactoring; it involves not only program refactorings, but also Feature Model (FM) refactorings, in order to improve configurability. However, FM refactorings are hard to define, due to the incompleteness of the refactoring catalogs developed as of today. In this paper, we propose a complete, sound catalog of algebraic laws, making up special FM refactorings that preserve configurability. This catalog is also defined as minimal, as one law cannot be derived from another one in the same catalog. In addition, a theory for FMs is presented, in the context of a theorem prover.
Key Words: feature models, refactoring, algebraic laws, software product lines
Category: D.2.2, D.2.7, D.2.13

1 Introduction

Software Product Lines (SPL) [Clements and Northrop 2001] may be adopted by either bootstrapping existing software products into a SPL (extractive approach), or extending an existing SPL to encompass an additional software product (reactive approach). Also, there can be a combination of both [Clements and Northrop 2001]. Extractive and reactive approaches can be enacted by the application of program refactorings. However, the definition of program refactoring [Fowler 1999] does not take into account intrinsic characteristics of SPL, such as feature models (FM) [Czarnecki and Eisenecker 2000], discussed in Section 3. For instance, using program refactorings in a SPL may cause

a reduction its configurability (decreasing the number of possibly-generated software products in a SPL), which should not happen in practice.

The term refactoring was coined by Opdyke in his thesis [Opdyke 1992]. He proposes refactorings as behavior-preserving program transformations in order to support the design of object-oriented application frameworks in an iterative manner [Opdyke 1992]. The cornerstone of his definition is that refactorings must imply in correct compilation of the refactored program and maintenance of its observable behavior. Opdyke's work and many of the later progresses in refactoring can be applied to frameworks (a technology heavily used today in SPL development), often introducing variation points. Nevertheless, as *program transformations*, refactorings do not handle configurability-level issues, which are only addressable at the FM level, nor do they define extractive transformations from two or more existing applications into a SPL. For instance, using program refactorings in a SPL may have the undesirable effect of reducing its configurability (instances of a SPL), which is not useful in practice. In this context, The notion of SPL refactoring [Alves et al. 2006] is an extension of the traditional definition of refactoring; it involves not only program refactorings, but also Feature Model (FM) refactorings, in order to improve configurability.

A FM refactoring is sound when it transforms a FM by improving (maintaining or increasing) its configurability. New refactorings are usually proposed by refactoring designers, by defining general transformations or templates. Developers can use them in a refactoring process, in this case a specific transformations. A catalog containing a number of general FM refactorings, which maintain and increase configurability, has been proposed [Alves et al. 2006]; however, the catalog presented is not proved to be complete and minimal. Here we present all refactorings. First, it is likely that refactoring designers will have to *increase the catalog*. Otherwise, *ad hoc* refactoring may be necessary, an error-prone and time consuming activity. Nevertheless, *it is hard for refactoring designers to propose new sound FM refactorings*. Checking soundness either with or without the help of a theorem prover [Gheyi et al. 2006a] requires extra expertise.

We propose, in this article, a complete set of sound algebraic laws for FMs (Section 5). An algebraic law is a FM refactoring that is guaranteed to preserve configurability. Moreover, we prove that this catalog is minimal, since there is no algebraic law that can be derived from another law. The laws define primitive transformations; we can derive interesting and useful coarse-grained refactorings by their composition. This benefit is illustrated through an example. If two FMs present the same configurability, the developer can always relate them by using our catalog, whose laws are based entirely on syntactic conditions. Therefore, there is no need for developers to do semantic reasoning, making their task more productive.

We also present a theory for FMs using the Prototype Verification System

(PVS) [Owre et al. 1998] (Section 2). PVS encompasses both a formal specification language and a theorem prover. The transformations and the completeness result are proven sound with respect to this theory within the prover (Section 4).

2 PVS Overview

The Prototype Verification System (PVS) provides mechanized support for formal specification and verification [Owre et al. 1998]. The PVS system contains a specification language and a theorem prover. Each specification consists of a collection of theories. Each theory may introduce types, variables, constants, and may introduce axioms, definitions and theorems associated with the theory. Specifications are strongly typed, meaning that every expression has an associated type.

Suppose that we want to model part of a banking system in PVS, on which each bank contains a set of accounts, and each account has an owner and a balance. Next, we declare two uninterpreted types (`Bank` and `Person`), representing sets of banks and persons, and a record type denoting an account. An uninterpreted type imposes no assumptions on implementations of the specification, contrasting with interpreted types such as `int`, which imposes all axioms of the integer numbers. Record types, such as `Account`, impose an assumption that it is empty if any of its components types is empty, since the resulting type is given by the cartesian products of their constituents. The `owner` and `balance` are fields of `Account`, denoting the account's owner and its balance, respectively.

```
Bank: TYPE
Person: TYPE
Account: TYPE = [# owner: Person, balance: int #]
```

In PVS, we can also declare function types. Next, we declare two functions types (mathematical relation and function, respectively). The first one just declares its name, parameters and result types, establishing that each bank relates to a set of accounts. The second function not only declares the withdraw operation, but also defines the associated mapping.

```
accounts: [Bank -> 𝒫[Account]]
withdraw(acc: Account, amount: int): Account =
  acc WITH [balance := (balance(acc)-amount)]
```

The `balance(acc)` expression denotes the balance of the `acc` account. The `WITH` keyword denotes the override operator, which replaces the mapping for `acc` by a new tuple, if `acc` is originally in the function domain. In the `withdraw` function, the expression containing the `WITH` operator denotes an account with the same owner of `acc`, but with a balance subtracted of `amount`. Similarly, we can declare a function representing the credit operation.

Besides declaring types and functions, a PVS specification can also declare axioms, lemmas and theorems. For instance, next we declare a theorem stating that the balance of an account is not changed when performing the withdraw operation after the credit operation with the same amount.

```
withdrawCreditTheorem: THEOREM
  ∀ (acc: Account, amount: int) :
    balance(withdraw(credit(acc,amount),amount)) = balance(acc)
```

3 Feature Models

In this section, we provide an overview of FMs. A FM represents the common and variable features of a SPL and the dependencies between them [Czarnecki and Eisenecker 2000]. A feature diagram is a tree-like 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 – represented by an unfilled circle), *Mandatory* (features that are required – represented by an filled circle), *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.

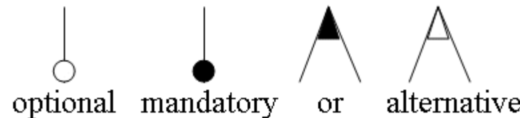


Figure 1: Feature Diagram Notations

Besides these relationships, FMs may include propositional logic formulae 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*, $mp3$ 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*.

The semantics of a FM is the set of its possible (valid) configurations (software product). A configuration contains a set of feature names; if *valid*,

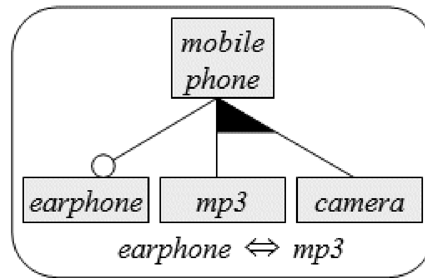


Figure 2: Feature Model Example

it satisfies all constraints (relations and formulae) 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.

4 Theory for Reasoning about Feature Models

In this section, a theory for formally reasoning about FMs is presented. We propose an abstract syntax in Section 4.1. Section 4.2 presents semantics for this language. In this paper, we use this theory to formally prove algebraic laws for FMs.

4.1 Abstract Syntax

Next we show the abstract syntax of our FM language in PVS. Hereafter, we use well-known mathematical symbols instead of PVS keywords, such as `SET`, `AND` and `FORALL`, for improving readability. A feature model (`FM`) contains a root and a set of feature names, relations and formulae. A `Name` is represented by a PVS type.

```
Name: TYPE
FM: TYPE = [# root: Name,
              features: 𝒫[Name],
              relations: 𝒫[Relation],
              formulae: 𝒫[Formula] #]
```

Our FM language contains six kinds of propositional formulae: true, false, feature name, negation, conjunction and implication.

```
form ::= true | false | feature | ¬ form |
         form ∧ form | form ⇒ form
feature ::= id
```

Note that we can derive other propositional formulae using the previous basic formulae (`f∨g = ¬((¬f)∧(¬g))`).

We represent formulae and the four relations with abstract datatypes [Owre et al. 1998]. In the following fragment, we specify all FM formulae considered.

```
Formula: DATATYPE
BEGIN
  TRUE_FORMULA: TRUE?: Formula
  FALSE_FORMULA: FALSE?: Formula
  NAME_FORMULA(n: Name): NAME?: Formula
  NOT_FORMULA(f: Formula): NOT?: Formula
  AND_FORMULA(f0, f1: Formula): AND?: Formula
  IMPLIES_FORMULA(f0, f1: Formula): IMPLIES?: Formula
END Formula
```

A PVS datatype is specified by providing a set of *constructors* (`NAME` and `AND`) along with associated *accessors* (`n` and `f`) and *recognizers* (`AND?`). For instance, `AND?(form)` is true when `form` is a conjunction formula. When a datatype is type checked, a new theory is created with the axioms and induction principles needed to ensure that the datatype is the initial algebra defined by the constructors [Owre et al. 1998].

The subsequent fragment represents the four FM relations using a PVS abstract datatype. The `p`, `c`, `l` and `r` denote the parent and child features, and the left and right subfeatures of a parent feature, respectively.

```
Relation: DATATYPE
BEGIN
  OPT_REL(p,c: Name): OPT?: Relation
  MAND_REL(p,c: Name): MAND?: Relation
  ALT_REL(p,l,r: Name): ALT?: Relation
  OR_REL(p,l,r: Name): OR?: Relation
END Relation
```

It is important to mention that an alternative and *or* relations can be between more than two subfeatures. We can specify them similarly. However, for simplicity, we only consider two subfeatures.

4.2 Semantics

In this section, we present the semantic notion for FMs. Our aim is to encode semantics for feature models in PVS in order to use its prover to prove properties about feature models, as we show in Section 5.

Each feature model defines a set of valid configurations, each of which containing a set of selected features, represented by `Configuration`. A valid configuration is a selection of features from the model satisfying its relations (implicit constraints) and formulae (explicit constraints), as declared next.

```
Configuration: TYPE = [# value: 𝒫[Name] #]
semantics(fm: FM): 𝒫[Configuration] =
  {c:Configuration | satImpConsts(fm,c) ∧
                     ∀ f:forms(fm) | satFormula(f,c)}
```

So, the semantics of a feature model is the set of all valid configurations. The expression `value(c)` yields all selected features in the configuration `c`. The predicate `satImpConsts(fm,c)` checks not only whether `c` contains a subset of `fm` features, but also it verifies whether the FM root is selected in `c`. It also checks whether `c` satisfies all relations of `fm`.

```
satImpConsts(fm: FM, c:Configuration): boolean =
  value(c) ⊆ features(fm) ∧
  root(fm) ∈ value(c) ∧
  ∀ r: relations(fm)| satRelation(r, conf)
```

The predicate `satRelations(fm,c)` checks whether the configuration `c` satisfies all relations of `fm`. For example, `c` satisfies a mandatory relation between features `A` and `B` when both features are selected in `c`. Next specify it in PVS.

```
satRelation(r:Relation, conf:Configuration): boolean =
  CASES r OF
    OPT_REL(p,c):
      c ∈ value(conf) ⇒ p ∈ value(conf),
    MAND_REL(p,c):
      p ∈ value(conf) ⇔ c ∈ value(conf),
    ALT_REL(p,l,r):
      l ∈ value(conf) ⇒ ¬ r ∈ value(conf) ∧
      r ∈ value(conf) ⇒ ¬ l ∈ value(conf) ∧
      p ∈ value(conf) ⇒ (l ∈ value(conf) ∨ r ∈ value(conf)) ∧
      l ∈ value(conf) ⇒ p ∈ value(conf) ∧
      r ∈ value(conf) ⇒ p ∈ value(conf),
    OR_REL(p,l,r):
      p ∈ value(conf) ⇒ (l ∈ value(conf) ∨ r ∈ value(conf)) ∧
      l ∈ value(conf) ⇒ p ∈ value(conf) ∧
      r ∈ value(conf) ⇒ p ∈ value(conf)
  ENDCASES
```

Next we present when a configuration satisfies a formula (`satFormula`). It has the standard semantics. For instance, a configuration satisfies the conjunction formula `p∧q` if it satisfies `p` and `q`. As another example, a configuration `c` satisfies the feature name formula `n` if `n` is a value selected in `c`.

```
satFormula(f:Formula, c:Configuration): RECURSIVE boolean =
  CASES f OF
    TRUE_FORMULA: TRUE,
    FALSE_FORMULA: FALSE,
```

```
      NAME_FORMULA(n): n ∈ value(c),
      NOT_FORMULA(f1): ¬ satFormula(f1,c),
      AND_FORMULA(f1, f2): satFormula(f1,c) ∧ satFormula(f2,c),
      IMPLIES_FORMULA(f1, f2): satFormula(f1,c) ⇒ satFormula(f2,c)
    ENDCASES
  MEASURE complexity(f)
```

Notice that `satFormula` is a recursive function. The PVS recursive functions must be declared using the `RECURSIVE` keyword. Moreover, a *measure function* [Owre et al. 1998] (in our case `complexity`), which is a well-founded order relation, must be specified in order to show the recursive function is total.

From the `semantics` definition, we proved in PVS a theorem stating that if two feature models have the same semantics, they contain the same *selectable feature names*. *Dead features* are excluded. For instance, if we deduce a formula ¬ `n` from the model, a feature `n` is not a selectable feature in a feature model. Although this situation is rare, it must be considered when doing formal reasoning. So, we only consider all selectable features in the following theorem.

Theorem Same Names *If arbitrary feature models* $m1$ *and* $m2$ *have the same semantics (semantics(m1)=semantics(m2)), they must have the same selectable features: selectable(m1) = selectable(m2).*

The *selectable* relation yields all selectable features from a FM. The previous theorem is important in the completeness result of our catalog. The proof of this theorem can be found in our technical report [Gheyi et al. 2006a].

5 Algebraic Laws

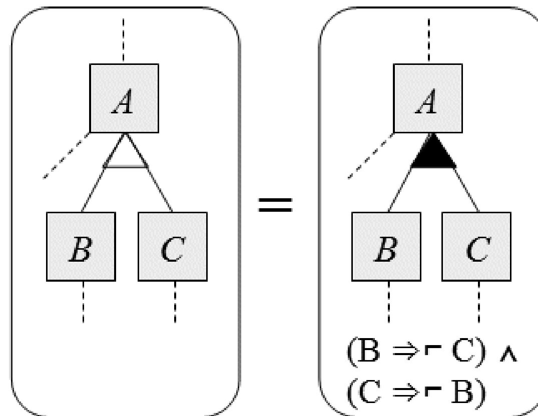
In this section, we propose a complete catalog of algebraic laws, which are a special kind of FM refactoring that preserves configurability. Each law is a *bidirectional sound refactoring*. In Section 5.1, we apply our algebraic laws to refactor the feature model depicted in Figure 2. We use our theory (Section 4) to prove them sound in PVS in Section 5.2. In order to prove the completeness result (Section 5.3), we present a reduction strategy stating how to convert any feature model to propositional logic formulae.

First we explain the notation used to depict the transformations. Each *algebraic law* consists of two *templates* (patterns) of FMs, on the left-hand (LHS) and right-hand (RHS) sides. We can apply a law whenever the left template is matched by a 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 the 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. An arrow on the top of a feature indicates that this feature is the root of the feature model. fs and $forms$ denote a set of features and formulae, respectively.

The first law relates the alternative and *or* relations. Applying Law 1 from left to right allows us to convert an alternative relation into an *or* relation along with two formulae establishing equivalent constraints. Similarly, by applying the law from right to left, we can convert an the *or* to an alternative relation.

Law 1 ⟨*replace alternative*⟩



Law 2 replaces an or relation and optional nodes. Laws 1 and 2 can be applied when there is more than two child features, as well.

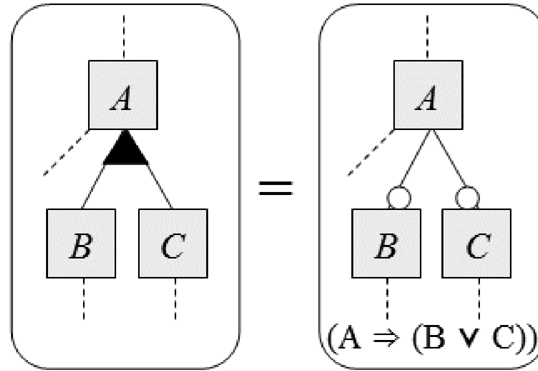
Law 3 relates a mandatory feature with an optional feature with a formula stating the same fact. Law 4 removes an optional feature and states the same fact in a formula. The root of a feature model always appears in all valid configurations. We have another law (Law 5) that removes a root and includes a formula stating that the root is always present.

Law 6 removes a feature that can never be selected. The $B \rightarrow false$ notation within brackets indicates that all occurrences of B in $forms$ are replaced by $false$. Since B cannot be selected, it is represented by $false$. We have to remove all occurrences of B in order to preserve well formedness. Similarly, this law allows us to add a set of nodes if we add a formula stating that the nodes cannot be selected.

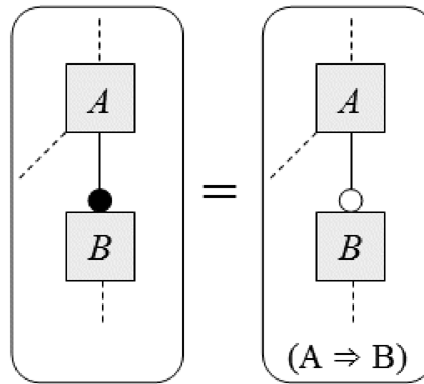
Finally, Law 7 allows us to add or remove formulae deducible from the model. Since it is a deducible formula, the semantics is preserved.

Some of the previous transformations may not be useful in practice since they convert a well-formed feature model to another that is not a tree, such as

Law 2 ⟨*replace or*⟩



Law 3 ⟨*replace mandatory*⟩



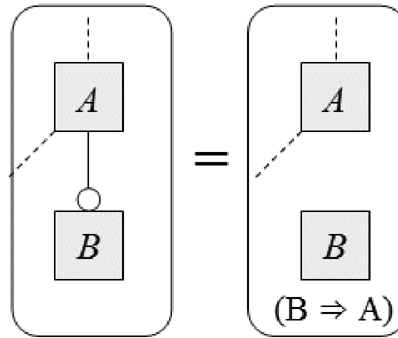
Law 4. However, they are very important in the theoretical reasoning, and in some intermediate refactoring steps, such as the one presented in Section 5.1.

5.1 Refactoring Example

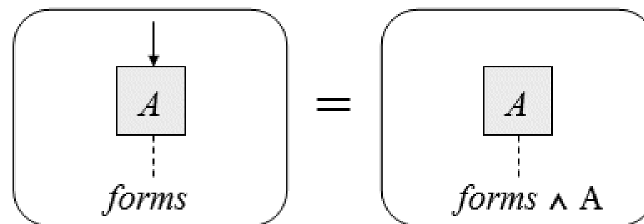
In this section, we apply our algebraic laws to the feature model depicted in Figure 2 in order to change its structure. Notice that we have a propositional formula stating that the *mp*3 feature is selected if and only if the *earphone* feature is selected. This constraint means that if the *mp*3 feature is selected, then the *earphone* feature is mandatory. Suppose that we would like to change the structure of the FM in Figure 2 to express this intuition.

Our aim is to change the parent of *earphone* to be *mp*3 using the mandatory relationship. First we apply Law 4 from left to right in order to remove the

Law 4 ⟨*remove optional*⟩



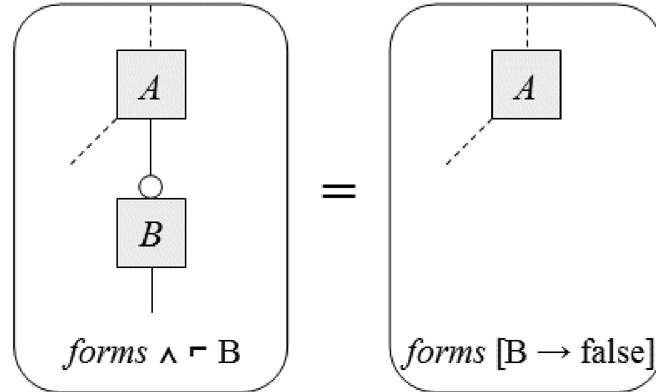
Law 5 ⟨*remove root*⟩



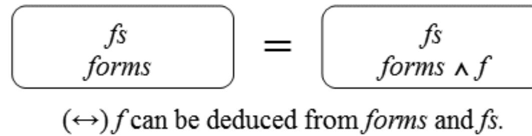
optional feature *earphone*. We state the same fact in a formula. Notice that this intermediate step yields a ill-formed FM since *earphone* is not connected to the tree. By applying Law 7, we can introduce some deducible formulae, which are important to reintroduce the *earphone* optional feature using Law 4 from right to left. The intermediate FM becomes well-formed again. Now the parent of *earphone* is *mp*3. Finally, we can introduce a mandatory between the *earphone* and *mp*3 features using Law 3 from right to left. The resulting model contains a deducible formula that can be removed by applying by applying Law 7. The entire refactoring process is summarized in Figure 3.

Notice that we apply small-grained transformations and this activity may be time consuming. We propose primitive transformations in order to have a complete and minimal set of algebraic laws. However, the main goal of our set of laws is to compose them to derive interesting coarse-grained refactorings. Since each primitive law is sound, we derive sound refactorings by composing them.

Law 6 ⟨*remove node*⟩



Law 7 ⟨*add formula*⟩



(↔) *f* can be deduced from *forms* and *fs*.

5.2 Soundness

We used the theory presented in Section 4 to prove that each law presented before preserves configurability. Next we present our approach to prove algebraic laws in PVS. The following theorem ensures that a given law preserves configurability (semantics).

```
law: THEOREM
  ∀ m1,m2:FM ... |
    syntax(...) ∧ conditions(...) ⇒ semantics(m1) = semantics(m2)
```

The predicates `syntax` and `conditions` describe the syntactic similarities and differences between the LHS and RHS models in the law, and conditions of the transformation. In general, we map each construction in the law to each corresponding element in FM semantics. Since all laws are primitive and have syntactic conditions, both predicates are easily defined. Proving a transformation increased our confidence in situations that a transformation did not preserve semantics. The theorem prover gave us insights on details of this formalization.

For example, next we specify the PVS theorem used to prove Law 4. The LHS and RHS FMs are represented by the `abs` and `con` variables. In general, for each element in the law, we declare a variable for it in our PVS theory. For

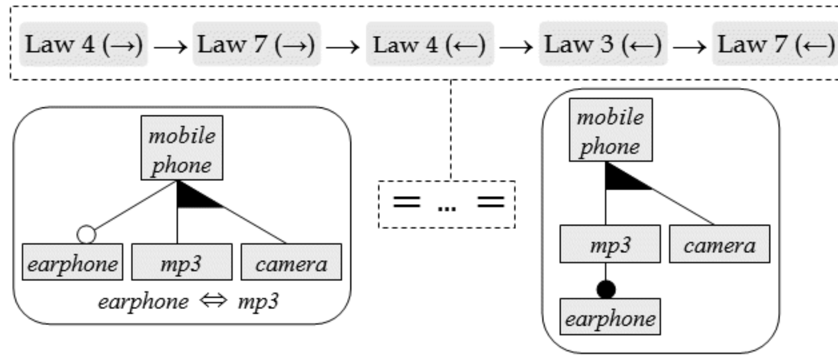


Figure 3: Feature Model Refactoring

instance, the meta-features `A` and `B` are represented by the PVS variables `A` and `B`, respectively. The optional relation on the LHS model is represented by `r`. Finally, the formula on the RHS model is represented by `f`. Notice that this law does not have conditions. Therefore, the `conditions` predicate is omitted.

```
removeOptional: THEOREM
  ∀ abs,con:FM, A,B:Name, r:Relation, f:Formula |
    syntaxRemOpt(abs,con,A,B,r,f) ⇒
      semantics(abs) = semantics(con)
```

Now we specify the `syntaxRemOpt` predicate relating both FMs. First of all, both FMs have the same root and features. The LHS model contains an optional relation `r` that is not declared in `con`. Finally, the RHS model contains a formula `f` that is not declared in `abs`.

```
syntaxRemOpt(abs,con:FM, A,B:Name, r:Relation, f:Formula): boolean =
  root(abs) = root(con) ∧
  features(abs) = features(con) ∧
  relations(abs) = {r1:Relation | r1 ∈ relations(con) ∨ r=r1} ∧
  r ∉ relations(con) ∧
  r = OPT_REL(A,B) ∧
  formulae(con) = {f1:Formula | f1 ∈ formulae(abs) ∨ f1=f} ∧
  f ∉ formulae(abs) ∧
  f = IMPLIES_FORMULA(NAME_FORMULA(B),NAME_FORMULA(A))
```

The `removeOptional` theorem was then proved using the PVS prover.

5.3 Completeness and Minimality

In this kind of work, it is important to evaluate whether the catalog is expressive enough to derive a representative set of transformations. Moreover, it is important to have a minimal set of transformations since increasing the catalog may

be difficult for developers to know all transformations and in which situations to apply them. In this section, we prove that our catalog of algebraic laws is complete and minimal.

As mentioned before, each law defines two transformations. The catalog is minimal since each law deals with one different (orthogonal) construct each time, and defines primitive and small-grained transformations. Therefore, one transformation cannot be derived from another.

Theorem Minimality *No feature model transformation stated by Laws 1-7 can be derived from any set of the others.*

We have defined one law for each construct of the feature model language. If two feature models $m1$ and $m2$ are equivalent (both have the same semantics), we can always reduce one model to another by applying Laws 1-7, as stated by the following theorem.

Theorem Completeness *If two feature models have the same semantics, we can always relate them applying Laws 1-7.*

In order to prove the completeness theorem, first we apply our laws in order to reduce any FM model to its equivalent one expressed in the propositional logic (*reduction strategy*). Figure 4 illustrates our reduction strategy.

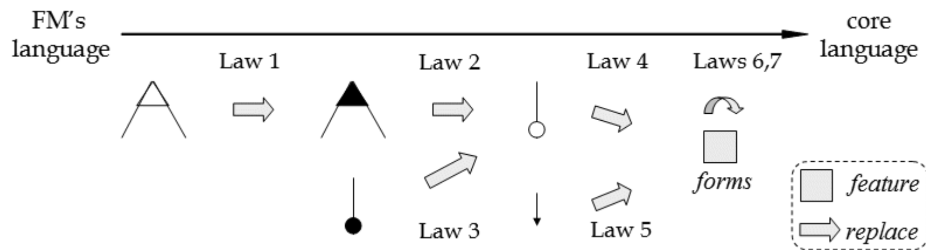


Figure 4: Reduction Strategy

We can reduce any FM to propositional logic by applying Laws 1-6 from left to right in this order. First we apply Law 6 from left to right in order to remove all features of `m1` that cannot be selected. The resulting model is `m1'`. Then we apply Laws 1-5 from left to right in order to remove all syntactic sugar constructs of `m1'`. This step reduces `m1'` to an equivalent model in the propositional logic. Notice that we can always apply them since there is no precondition. The

resulting model is represented by its components (names1, forms1). We only have feature names and formulae at this stage. This reduction always *terminates* and *converges* since each law does not introduce another construct that was previously removed. Notice that the number of FM relations is decreased in each step.

After applying the reduction strategy, now our aim is to show that the two FMs in the propositional logic are equivalent. If two feature models have the same semantics, they have the same selectable feature names, hence names1=names2 (Theorem Same Names). Both models may have different formulae. Since m1 and m2 have the same meaning, forms1 and forms2 are equivalent. Additionally, propositional logic calculus is a complete calculus. Therefore, we can always prove that forms1=forms2 by applying Law 7. Now the resulting model is (names2, forms2). Finally, we reconstruct m2 by adding all syntactic sugar constructs (by applying Laws 1-5 from right to left) and all feature nodes that can never be selected by applying Law 6 from right to left.

The detailed proof of the completeness result is presented next. For simplicity, we consider alternative and mandatory relations containing two child features; the proof is similar for FMs containing more than two child features. Notice that all proposed algebraic laws are used in the proof. This is another evidence of the minimality result. Every step in the following derivation is justified within brackets.

```
 m1
   [applying Law 6 (→) to all
    features that cannot be selected]
= m1'
   [applying Laws 1-5 (→) until
    there is no syntactic sugar construct]
= (names1, forms1)
   [semantics(m1) = semantics(m2),
    Theorem Same Names]
= (names2, forms1)
   [semantics(m1) = semantics(m2),
    propositional logic calculus is complete,
    applying Law 7]
= (names2, forms2)
   [applying Laws 1-5 (←)
    including all syntactic sugar constructs]
= m2'
   [applying Law 6 (←)]
= m2
```

6 Related Work

Deursen [van Deursen and Klint 2002] proposes a textual language for describing features. Their language is similar to ours, but it does not consider formulae.

Their semantics is equivalent to ours. Also, a set of fifteen rules relating equivalent FMs are proposed, which are very similar to our algebraic laws. They informally argue soundness, in contrast to our approach, which uses PVS to increase confidence. However, Deursen does not propose a minimal and complete set of FM algebraic laws.

Benavides et al. [Benavides et al. 2005] propose 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 using our theory in PVS. Their idea of filters is equivalent to formulae in our FMs.

Batory [Batory 2005] integrates prior results to connect feature diagrams, grammars, and propositional formulae. This connection also allows the use of SAT solvers to help debug feature models by confirming compatible and incomplete feature sets. In our work we propose semantics for FMs, and propose a complete and minimal set of algebraic laws for feature models.

Liu et al. [Liu et al. 2006] proposes Feature Oriented Refactoring (FOR), which is the process of decomposing a program, usually legacy, into features. Such work focuses on configuration knowledge, specifying the relationships between features and their implementing modules, backed by a solid theory. Also, the authors present a semi-automatic refactoring methodology to enable the decomposition of a program into features. In our work, we focus on refactorings at a different level (models instead of programs).

Czarnecki et al. [Czarnecki et al. 2005] introduce 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. Antkiewicz and Czarnecki [Antkiewicz and Czarnecki 2004] present a FeaturePlugin, which is a feature modeling plug-in for Eclipse. The tool supports cardinality-based [Czarnecki et al. 2005] 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 by proving a theorem in PVS. However, our theory does not handle cardinality-based FMs. We can check them if we extend our theory.

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 they do not con-

sider formulae different from our approach. Moreover, the previous approaches do not prove meta-properties as in our work. Our laws define meta-transformations. The previous approaches focus on proving properties of a specific FM.

Our prior work [Gheyi et al. 2006b] proposes another theory of FMs in Alloy [Jackson 2006]. We can perform analysis on Alloy models using the Alloy Analyzer tool. In both encodings we can check whether a transformation is a refactoring. However, Alloy is not a theorem prover, differently from PVS. It would be nice to combine model checking and theorem proving. We proposed [Alves et al. 2006] a catalog containing a number of general FM refactorings (which preserve or increase configurability). However, this catalog is not proved to be complete. In this work, we propose a sound, complete and minimal catalog of algebraic laws for FMs. Moreover, we mechanize a FM theory in PVS.

7 Conclusions

In this article, we propose a sound, complete and minimal catalog of algebraic laws for Feature Models (FMs). The laws are proven sound with respect to a formal semantics as specified in PVS. Our PVS theory can be used by others who wish to prove properties about FMs. Our catalog can be used when refactoring Software Product Lines (SPLs). As previously explained, besides traditional program refactoring, FMs must also be refactored. This task brings additional benefit, since it improves the quality of FMs by preserving or increasing its configurability. Therefore, developers do not need to reason based on semantics in order to refactor a FM, as the catalog can be directly applied.

Our reduction strategy reduces every possible FM to propositional formulae. As an alternative, only one law to introduce deducible formula could be proposed in order to restructure FMs, presenting a single semantic condition. However, it is much easier to deal with graphical (syntactic) notations in SPL refactorings instead of propositional formulae. Most refactorings can be performed by only checking syntactic conditions, which a supporting tool could easily incorporate. As a result, we propose laws for the graphical notation.

As future work, theory can be extended to include additional constraints, such as cardinality FMs. Based on this extended theory, we can propose and proving new algebraic laws. Furthermore, we focus on configurability-preserving transformations. We intend to propose a complete catalog of FM refactorings increasing configurability. Finally, we can specify well-formedness rules for FMs in PVS, and prove that our laws preserve them.

Acknowledgements

We thank the SBLP anonymous referees for the useful suggestions. Gheyi and Massoni's work was partially done at Federal University of Pernambuco. We

would like to thank all members of the Software Productivity Group (SPG) at UFPE, for their important comments. This work was supported by CNPq (Brazilian research agency).

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.

[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.

[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.

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

[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.

[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.

[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.

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

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

[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.

[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.