Generative Programming And Component Engineering, 2009

Safe Composition of Non-Monotonic Features

Martin Kuhlemann
Faculty of Computer Science
University of Magdeburg, Germany
mkuhlema@ovgu.de

Don Batory
Department of Computer Sciences
University of Texas at Austin, USA
batory@cs.utexas.edu

Christian Kästner
Faculty of Computer Science
University of Magdeburg, Germany
ckaestne@ovgu.de

Abstract

Programs can be composed from features. We want to verify automatically that all legal combinations of features can be composed safely without errors. Prior work on this problem assumed that features add code monotonically. We generalize prior work to enable features to add *and remove* code, describe our analyses and implementation, and review case studies. We observe that more expressive features increase the complexity of developed programs rapidly – up to the point where tools and automated concepts as presented in this paper are indispensable for verification.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; D.2.13 [*Software Engineering*]: Reusable Software; D.2.1 [*Software Engineering*]: Requirements/Specifications

General Terms Verification, design

Keywords Feature-oriented programming, safe composition, refactoring, AHEAD

1. Introduction

In feature-oriented programming, *features* encapsulate increments in program functionality [9]. Features can be implemented as code transformations called *feature modules* [9, 1]. Composing feature modules in different ways yields a family of programs called a *product line* [30].

Not all combinations of features are meaningful [8]. Meaningful combinations are legal to a *feature model* which declares features to be mandatory, optional, alternative, or inclusive to other features [18]. Unfortunately, developers cannot verify properties of *all* programs in a product line by simply composing and analyzing every program in isolation (brute force strategy) as the number of programs can be exponential in the number of features [22]. One solution to this problem is *safe composition*, a technique to verify that all programs of a product line that are assembled from feature modules are type correct [11, 35, 21, 20]. Safe composition analyses effectively and efficiently analyze all compositions of features. In prior work, safe composition approaches have been proposed for *monotonic* feature modules that could add new classes to a program, add new members to existing classes, and wrap existing methods. Members or classes could never be deleted by a feature.

We recently proposed that feature modules be extended in a fundamental way: to include object oriented refactorings [23], which can rename, add, and delete existing classes and members. With refactorings, feature modules are now more expressive but also break with the common assumption of monotonicity [5] where code elements can only be added or extended. For example, a feature can now rename a class or method M to N. If a subsequently added feature references M, the resulting program is no longer correct and will not compile. The benefits of adding refactorings to feature modules is not the subject of this paper (this is explored elsewhere [23]), but we focus on the analyses to support refactorings as elements of feature modules.

In this paper, we show how safe composition analyses can be generalized to allow features to be non-monotonic transformations (including but not limited to refactorings). To show that the resultant computational effort is manageable for non-trivial programs, we present a number of case studies.

We observe that when feature transformations can create and delete code, the complexity of developed programs can grow rapidly, more than we initially expected. We reason that a more expressive language can increase the complexity of developed programs rapidly to the point where tools and automated concepts as presented in this paper are indispensable for verification.

2. Background

2.1 Safe Composition in Feature-Oriented Design

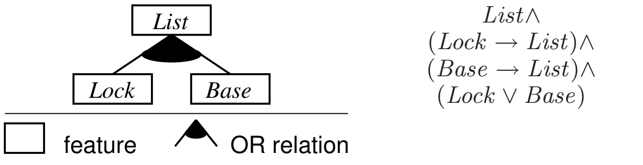
Feature models. A feature is an increment in program functionality [9]. A feature is implemented by a sequence of primitive transformations (i.e., add method, add field, wrap method, etc.) called a feature module. Feature modules are selected during a configuration process to define a target program. When a feature module is selected, its transformations are applied to that program.

Meaningful combinations of features are defined in a feature model [18]. A feature model additionally defines the *order* in which selected feature modules are composed [8].

A feature model is depicted by a feature diagram like the one in Figure 1a. This model defines an abstract data type List with the features *Lock* and *Base* and maps these features to the feature modules *Lock* and *Base* respectively. The model says that any combination of *Lock* and *Base* yields a meaningful program, i.e., the legal *feature compositions* according to this feature model are *Lock*, *Base*, and *Lock* • *Base* (denoting *Lock* applies transformations to *Base*). We encode the feature composition order in the feature model by reading from right to left in the feature diagram so *Base* • *Lock* (*Base* applies transformations to *Lock*) is not a legal program.

While the diagram notation is illustrative we need a different representation of feature models in this paper. We represent the model of Figure 1a as a propositional formula in Figure 1b following the standard rules in [11, 35, 7, 12]. Each variable in this

GPCE'09, October 4–5, 2009, Denver, Colorado, USA.



(a) diagram notation

$$List \wedge \\ (Lock \rightarrow List) \wedge \\ (Base \rightarrow List) \wedge \\ (Lock \vee Base)$$

(b) formula notation

Figure 1. Sample feature model.

formula corresponds to one feature – we thus call it a *feature variable*. A feature variable is ’true’ when the feature is selected and ’false’ otherwise. The formula evaluates to ’true’ for legal feature selections. Note, feature order is not defined in the formula because there is only one ordering. Feature order is defined externally to the formula notation.

Feature modules. Features are implemented by feature modules which successively transform the code contributions of previously composed features [31, 9]. They encapsulate classes and class refinements where class refinements add new members to classes or extend existing members. If a feature module is not selected then its classes and class refinements are not applied to the generated program.

In Figure 2, we show the feature modules referenced in the feature model of Figure 1 (we use the Jak language [9] which adds feature modules to Java). The module *Base* encapsulates a class List. The module *Lock* encapsulates a refinement of class List which adds a field _locked and a method setLock to class List of *Base*. Method get of the class refinement List (in *Lock*) refines method get of the class List (in *Base*) by wrapping. It adds statements and calls the refined method using Jak’s keyword Super (Line 14). Since Jak feature modules only add code (classes, methods, fields, statements) we call them *monotonic modules*.

Safe composition of monotonic modules. The feature model in Figure 1 does not represent the set of type correct programs which can be composed from the referenced modules of Figure 2. Note that *Lock* requires the selection of *Base* for two reasons: (a) the class refinement of *Lock* requires the class List to exist and (b) method refinement List.get of *Lock* requires the refined method List.get to exist. Knowing this dependency we can infer that the program defined only by *Lock* is in error because *Base* is not selected, i.e., either the model is in error, feature module *Lock* is in error, or both. In this example, the feature model is in error.

Thaker et al. [35] determined dependencies, called *composition constraints*, between monotonic modules such that domain experts could attach them to the features in the feature model. These composition constraints restrict the legal compositions to those which compose without errors. To repair the feature model of Figure 1, a constraint can be added to require *Base* to be present each time *Lock* is selected – we use a propositional formula over feature variables to denote this constraint: $Lock \rightarrow Base$.

2.2 Refactoring Feature Modules

There are many use cases for which more expressive feature modules are beneficial, e.g., [4, 3, 23]. An interesting extension to monotonic feature modules is to allow them to create *and delete* code, i.e., an extension that breaks with common assumptions of monotonicity of feature modules. One example in which feature modules are extended to create and delete code is adding refactorings to feature modules.

A *refactoring* transforms a program by altering the program’s structure but not its semantics [29]. For example, modifying a method’s name and updating all references is a ’Rename Method’ refactoring [16]. The standard refactorings that we use to illustrate the concepts in this paper are ’Rename Method’ and ’Rename Class’ as we assume them to be widely known[1] – but, our approach is not limited to them.

Feature Module *Base*

```
 1  public class List {
 2    private MyList _elements;
 3    Object get(){
 4      return _elements.get(0);
 5    }
 6  }
```

Feature Module *Lock*

```
 7  refines class List {
 8    boolean _locked = false;
 9    void setLock(boolean newLock){
10      _locked=newLock;
11    }
12    Object get(){
13      if (_locked) return null;
14      return Super.get();
15    }
16  }
```

Figure 2. Sample feature-oriented design.

In previous work, we explored how refactorings could be included in feature modules called a *refactoring feature module (RFM)* [23]. Figure 3 shows RFMs in the order they can be applied sequentially to the feature module *Base* (top-down order). In line with sequentiality, an RFM only transforms code created in feature modules which precede the RFM in a composition but not code created after the RFM. For instance, RFM *ListAdt* in Figure 3 renames the class List to ADT of feature modules *Base* and *GetPop* when they are selected. Since RFMs create and delete code elements (e.g., renaming can be represented as code element deletion and creation), they are *non-monotonic*.

RFMs are beneficial for product lines of components [23]: A component generated from feature modules often needs to integrate with legacy applications [10], which expect particular names and signatures for a component’s interface, that is, applications which expect different names and signatures than those that are generated. RFMs can transform generated interfaces so that they can neatly be integrated with legacy code. RFMs in this use case are *final* transformations performed on generated components.

We focus on refactorings in RFMs that modify fully qualified names and signatures of code elements (’identifiers’ for short) because identifiers are important for module integration. Refactorings which depend on method bodies (e.g., ’Change Bidirectional Association to Unidirectional’[2]) are not considered but our concepts apply for them too once we analyze method bodies. Still, even without analyzing method bodies we cover implementations of about 28% of the refactorings in [16].[3]

[1] ’Rename Class’ changes the name of a class [16].

[2] ’Change Bidirectional Association to Unidirectional’ removes a field of a two-way association when it is not called/needed [16].

[3] Refactorings that we cover are: ’Add Parameter’, ’Change Unidirectional Association to Bidirectional’, ’Change Value to Reference’, ’Encapsulate Field’, ’Extract Class’, ’Extract Complete Interface’, ’Extract Superclass’, ’Hide Delegate’, ’Inline Method’, ’Introduce Parameter Object’, ’Move Class’, ’Move Field’, ’Move Method’, ’Rename Class’, ’Rename Field’, ’Rename Method’, ’Remove Assignments to Parameters’, ’Replace Constructor with Factory Method’, ’Replace Magic Number with Symbolic Constant’, ’Replace Method with Method Object’, ’Self Encapsulate Field’. Refactorings ‘Consolidate Duplicate Conditional Fragments’ and ’Replace Nested Conditional with Guard Clauses’ do not affect identifiers – thus, they are irrelevant for module integration and we count them as covered.

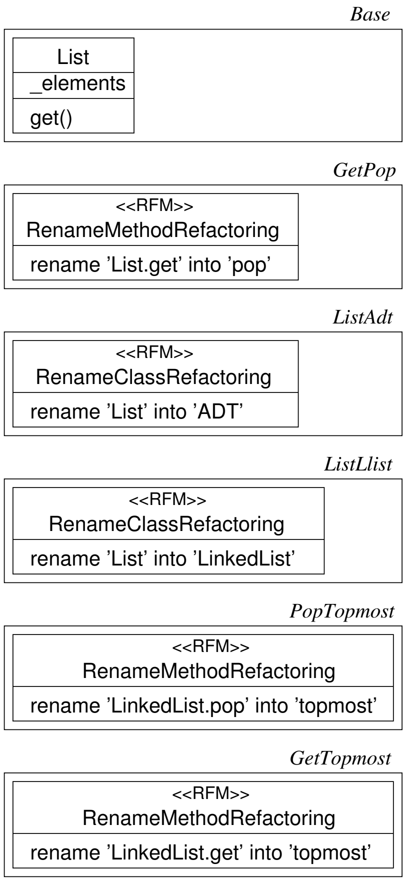


Figure 3. Refactorings as parts of feature modules.

Given the above, we want to verify that every legal combination of monotonic feature modules and non-monotonic RFMs that is permitted by a feature model is type correct. We examine the difficulty of doing so in the following sections.

3. Analysis of Non-Monotonic Modules

The key to safe composition of a non-monotonic feature module – and thus an RFM – is to verify its preconditions [32, 16, 29, 36]. For refactorings, preconditions are formulated in terms of identifiers which must exist when an RFM is applied and identifiers which must not exist. For instance, a refactoring which renames a class List into LinkedList requires both that a code element with identifier List to exist and that a code element with identifier LinkedList not to exist. If List does not exist then the refactoring fails because there is no class to rename. If LinkedList exists then the refactoring fails too as there can only be one LinkedList class in a program [29, 32]. If these constraints are fulfilled in all legal compositions then this 'Rename Class' RFM is guaranteed to compose safely.

To clarify the challenges, we deduce composition constraints (dependencies between features) for the refactorings of Figure 3. Note that the complexity of the constraints grows quickly as the number of RFMs increases:

- The 'Rename Method' refactoring in RFM *GetPop* renames method List.get into pop and thus requires a method with the identifier List.get. *Base* creates List.get and thus *GetPop* must always appear after *Base*. *GetPop* additionally requires that a code element with the identifier List.pop not to exist – this is always true as List.pop is not created by any combination of features prior to *GetPop*. The composition constraint for *GetPop* is that *GetPop* requires *Base* ($GetPop \rightarrow Base$).
- The 'Rename Class' refactoring in RFM *ListAdt* renames class List into ADT and thus requires that a code element with the identifier List to exist and a code element with identifier ADT not to exist. List is created by *Base* and ADT is not created by any feature that can be composed prior to *ListAdt*. Thus the composition constraint for *ListAdt* is $ListAdt \rightarrow Base$.
- The 'Rename Class' refactoring in RFM *ListLlist* renames class List into LinkedList and thus requires a code element with the identifier List and that there is no code element with the identifier LinkedList. Special to the constraint of *ListLlist* is feature *ListAdt*. *ListAdt* deletes List and so *ListLlist* can only be applied if *ListAdt* is not also selected. We derive a constraint $ListLlist \rightarrow (\neg ListAdt \wedge Base)$.
- The 'Rename Method' refactoring in RFM *PopTopmost* renames method LinkedList.pop into topmost and thus requires a code element with the identifier LinkedList.pop. LinkedList.pop can only be created by the feature composition $ListLlist \bullet GetPop \bullet Base$ *and* that feature *ListAdt not* be selected. The constraint is $PopTopmost \rightarrow (ListLlist \wedge \neg ListAdt \wedge GetPop \wedge Base)$.
- Finally *GetTopmost* renames method LinkedList.get into topmost and thus requires a code element with the identifier LinkedList.get and that there is no code element with identifier LinkedList.topmost. LinkedList.get can only be created by the feature composition $ListLlist \bullet Base$ and that features *GetPop* and *ListAdt* not be selected (here, this includes the non-existence of LinkedList.topmost): $GetTopmost \rightarrow (\neg Pop-Topmost \wedge ListLlist \wedge \neg ListAdt \wedge \neg GetPop \wedge Base)$

Note that the above constraints can be simplified. Example: $GetTopmost \rightarrow (ListLlist \wedge \neg GetPop)$.

Summary. In safe composition of monotonic feature modules, a feature *F* requires another feature *G* if *F* references a code element (e.g., method, field, class, interface) introduced in *G* [35]. In safe composition of non-monotonic RFMs, one RFM may require that multiple features apply in an ordered composition to create a code element with a particular identifier and may additionally require that certain features are *not* selected.

4. Safe Composition of Non-Monotonic Modules

In this section, we present our solution to verify safe composition of non-monotonic feature modules. We proceed in three steps: In Section 4.1 we describe the basic concept; in Sections 4.2 and 4.3 we describe the concepts of its implementation. We use the following notation: $\mathbb{F}$ denotes the set of all features, $\mathbb{C}$ denotes the set of all feature compositions (cf. Sec. 2.1) over $\mathbb{F}$, $\mathbb{P}$ denotes the set of all propositional formulas over feature variables[4], $\mathbb{I}$ denotes the set of all possible identifiers.

4.1 Basic Concept

Preconditions of refactorings define identifiers that either must exist or must not exist in the code for the refactoring to be successful. We determine composition constraints by relating feature compositions to each other that make the referenced identifiers exist (or not exist), i.e., where some code elements have these identifiers. We translate

[4] A feature variable is a propositional variable that is 'true' when the according feature is selected and 'false' otherwise (cf. Sec. 2.1).

these constraints to propositional formulas, as SAT solvers can verify them efficiently for all feature combinations.

To provide a concise syntax for the following discussions, we introduce a function p that translates feature compositions into propositional formulas ($p : \mathbb{C} \rightarrow \mathbb{P}$). The function p translates a composition of features into a conjunction of feature variables because all features in a composition must be selected in order to apply it. When features in a feature composition are not selected, their feature variables nevertheless contribute to the generated conjunction but are assigned as 'false'. As an example, p translates a feature composition $\mathit{ListAdt} \bullet \mathit{Base}$ (that does not select *GetPop*) into the formula $\mathit{ListAdt} \wedge \neg \mathit{GetPop} \wedge \mathit{Base}$, i.e., $p(\mathit{ListAdt} \bullet \mathit{Base}) = \mathit{ListAdt} \wedge \neg \mathit{GetPop} \wedge \mathit{Base}$. Features that are composed after *ListAdt* are not relevant.

Preconditions of refactorings fall into two camps: the existence of some identifiers I^+ and the non-existence of some identifiers I^- in the code to refactor ($\mathrm{I}^+ \subseteq \mathbb{I}$; $\mathrm{I}^- \subseteq \mathbb{I}$; $\mathrm{I}^+ \cap \mathrm{I}^- = \varnothing$). We create a composition constraint for each precondition. Both constraints must be fulfilled in every legal composition in order to compose a refactoring safely, i.e., every legal composition must make both identifiers in I^+ exist and identifiers in I^- not-exist immediately prior to that refactoring. Suppose a refactoring *R* requires some code element with the identifier x ($\mathsf{x} \in \mathrm{I}^+$): we verify that x always exists immediately prior to *R* in all legal feature compositions containing *R* in a product line. Further, suppose that a refactoring *R* requires some code element with the identifier y to not exist ($\mathsf{y} \in \mathrm{I}^-$): we verify that y never exists immediately prior to *R* in all legal feature compositions containing *R* in a product line.

We use the RFM *ListLlist* of Figure 3 as a running example in our discussions: *ListLlist* renames class List into LinkedList. Thus, *ListLlist* requires that List exist and LinkedList not exist in all legal compositions immediately before feature *ListLlist*.

Existence of identifiers. With a function c we calculate the set of unique feature compositions that make a code element with the required identifier x exist at the point immediately before the feature *R*. That is, $c : \mathbb{I} \times \mathbb{F} \rightarrow \mathcal{P}(\mathbb{C})$. For now, we assume that such a function exists, its efficient implementation is discussed later.

$$c(\mathsf{x}, R) = \{C_1, C_2, \ldots, C_n\} \tag{1}$$

For example, the set of compositions which make identifier List exist immediately before *ListLlist* is:

$$c(\mathsf{List}, \mathit{ListLlist}) = \{\mathit{Base}, \mathit{GetPop} \bullet \mathit{Base}\}$$

Consequently, a propositional constraint which we must verify for all legal feature compositions is: If feature *R* is selected then at least one composition in the result of $c(\mathsf{x},R)$ applies prior to it:

$$R \rightarrow (p(C_1) \vee p(C_2) \vee \ldots \vee p(C_n)) \tag{2}$$

For example, the composition constraint for *ListLlist* is[5]:

$$\mathit{ListLlist} \rightarrow (p(\mathit{Base}) \vee p(\mathit{GetPop} \bullet \mathit{Base}))$$

SAT solvers efficiently verify existential clauses for propositional formulas [27]. Therefore, we transform constraint (2) into a theorem to be verified by a SAT solver. That is, instead of verifying that every legal composition fulfills the composition constraint of feature *R*, we verify whether there is a composition that is legal for a feature model but that does not fulfill *R*'s constraint.

We use a key observation by Czarnecki [11]: if C is a constraint that is to be satisfied by programs in a product line and FM is the predicate that is derived from a feature model (like in Fig. 1b), then $FM \rightarrow C$ must be a tautology and thus $\neg(FM \rightarrow C)$ must be unsatisfiable. The theorem that we need to prove is that the following predicate is unsatisfiable:

$$\neg(FM \rightarrow (R \rightarrow (p(C_1) \vee p(C_2) \vee \ldots \vee p(C_n)))) \tag{3}$$

Formula (3) is unsatisfiable when *R* composes safely. Stated another way, if (3) is satisfiable then the binding provided by a SAT solver tells us a legal composition of features for which *R*'s precondition fails.

In our example, *ListLlist* can be composed safely if the following predicate is unsatisfiable. FM is the predicate derived from the feature model (not depicted but similar to Fig. 1b).

$$\neg(FM \rightarrow (\mathit{ListLlist} \rightarrow (p(\mathit{Base}) \vee p(\mathit{GetPop} \bullet \mathit{Base}))))$$

Non-existence of identifiers. We now consider how to verify the non-existence of an identifier. With our function c, we calculate the set of feature compositions that make y exist at the point immediately before *R* where *R* requires y to not exist.

$$c(\mathsf{y}, R) = \{C'_1, C'_2, \ldots, C'_n\} \tag{4}$$

For example, *ListLlist* in Figure 3 requires LinkedList to not exist – the set of compositions that create LinkedList immediately before *ListLlist* is $c(\mathsf{LinkedList}, \mathit{ListLlist}) = \varnothing$.

By symmetry, a constraint which we must verify for all legal feature compositions is whether feature *R* implies that no composition in the result of $c(\mathsf{y},R)$ applies prior to it:

$$R \rightarrow \neg(p(C'_1) \vee p(C'_2) \vee \ldots \vee p(C'_n)) \tag{5}$$

As before, we translate this constraint into a theorem for a SAT solver to verify. A binding that satisfies the formula tells us a legal composition for which *R*'s precondition fails (i.e., y exists).

In our running example, the result of $c(\mathsf{LinkedList}, \mathit{ListLlist})$ is the empty set and so the constraint becomes a tautology – thus the SAT test of the negated formula fails.

When the tests for either the existence or the non-existence of identifiers fail (a SAT-test succeeds) then we have found an error in the feature model, the feature modules, or both. That is, when feature *R* is selected, its precondition(s) can be violated. A domain expert is alerted and can now correct the feature-oriented design. If all constraints are verified, all programs of a product line can be safely composed.

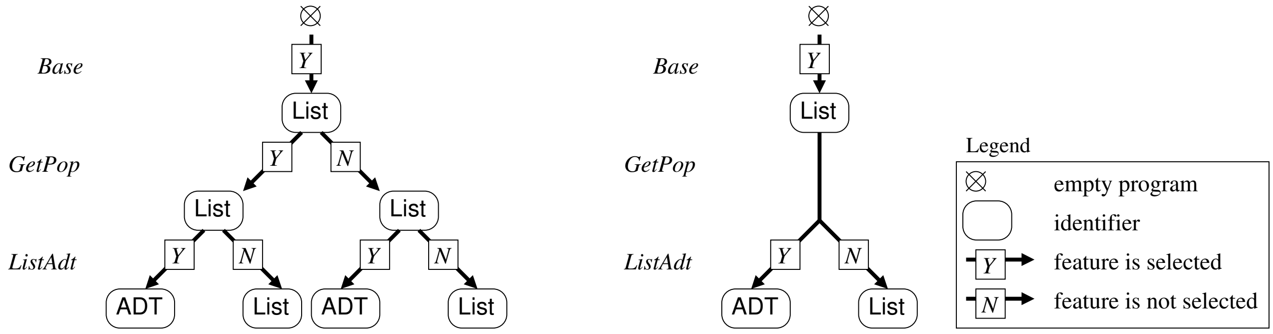
4.2 Computing Feature Compositions with Identifiers

In order to assemble composition constraints, we relied on a function *c* which determines compositions where some code element with a particular identifier exists. In this section, we show the concept how to implement *c* efficiently, i.e., how to calculate compositions without actually enumerating them all.

We record all identifiers of code elements that monotonic feature modules create and we establish each identifier as root of a decision tree. In the case where two features introduce two code elements with the same identifier then two root nodes (and thus two decision trees) emerge. The decisions recorded in these trees are whether features are selected or not. Feature decisions then map nodes of identifiers to nodes of new identifiers. A single decision tree represents one code element that has different identifiers over time (after RFMs have been applied). Decision trees may contain a special node indicating the code element got deleted (instead of transformed).

In Figure 4a, we visualize the effects of feature decisions prior to *ListLlist* from Figure 3 (*Base*, *GetPop*, and *ListAdt*) on class List

[5] $p(\mathit{Base}) = \neg \mathit{ListAdt} \wedge \neg \mathit{GetPop} \wedge \mathit{Base}$; $p(\mathit{GetPop} \bullet \mathit{Base}) = \neg \mathit{ListAdt} \wedge \mathit{GetPop} \wedge \mathit{Base}$; features composed after *ListADT* (predecessor of *ListLlist*) are not relevant



(a) unoptimized

(b) optimized

Figure 4. Decision trees for List of Fig. 3.

in *Base*. That is, we show the decision tree of List. In this tree, an arrow is a feature which transforms a code element's identifier (arrow-source) to a new identifier (arrow-target). With different labels for arrows, we indicate whether the decision to select a feature creates the child node or whether the decision to not select it does. Features that do not modify identifiers of code elements lead to the same identifier whether they are selected or not.

We iterate the features in a linear process, namely in the order that is defined in the feature model. In each iteration step we first verify safe composition for the current feature with the existing trees before we record the current feature's effects on identifiers in these trees, i.e., trees successively grow. Consequently, the trees we use to verify an RFM contain only effects of features that precede the currently verified RFM (as RFMs only transform code preceding features added/changed, cf. Sec. 2.2).

With decision trees for all identifiers we can determine compositions which include code elements with a particular identifier. When an RFM requires that a code element of a particular identifier exists immediately before the RFM then it requires a *leaf node* with this identifier in the decision trees (such leafs can exist in different trees[6]). The composition, that makes the tree's code element with its leaf's identifier exist before the verified RFM, emerges from the path of feature decisions from this leaf backward to the empty program.

To exemplify the use of decision trees in the function c, we recalculate the compositions prior to *ListLlist* that include List (in the prior section we assumed the function c to compute this set). That is, we want to calculate in Figure 4a the compositions that make List exist for *ListLlist*. From List leaf nodes in our decision tree we calculate two different compositions that contain List so $c(\mathsf{List}, \mathit{ListLlist}) = \{\mathit{Base}, \mathit{GetPop} \bullet \mathit{Base}\}$. We translate and use these compositions in order to create composition constraints and verify these constraints with SAT technologies.

As an aside, we do not need to encode orderings of features in propositional formulas because we defined that there is only *one* composition order for all features (cf. Sec. 2.1). This ordering is encoded in the trees which we create iteratively with that ordering. For example, when features *Base* and *GetPop* are both assigned 'true' by SAT, we know that that they apply in the order $\mathit{GetPop} \bullet \mathit{Base}$. Thereby, it does not matter whether $\mathit{GetPop} \wedge \mathit{Base}$ or $\mathit{Base} \wedge \mathit{GetPop}$ is assigned 'true'.

Special cases of individual refactorings require extensions to the above concepts. We omit their discussion here and focus on the core concepts for the clarity of this paper.

Optimization. The decision tree of every code element grows exponentially with the number of features it records. To reduce the size of decision trees, we do not add children to a node when both children have the same identifier as the current node, i.e., when the feature does not change the identifier of a code element. For example, the 'Rename Method' RFM *GetPop* does not change the identifier of class List. Thus, *GetPop* adds two children to node List in Figure 4a but both have the same identifier as their father node. In the optimized trees, we do not add either child to the List node when recording *GetPop*. The optimized decision tree for List is shown in Figure 4b. In practice, almost all monotonic feature modules and RFMs will not change a given identifier of a code element (see case studies Sec. 5) and this makes it an important optimization.

Optimized decision trees allow us to compute *patterns* for compositions with a code element of a particular identifier when we calculate paths from leafs to the empty program. Inside such a pattern only features appear which decide whether a code element with this identifier exists, e.g., *GetPop* is not in the pattern of List (cf. Fig. 4b; *GetPop* is not in the path of decisions from leaf List towards the empty program) because *GetPop* does not decide the existence of List. We use these patterns instead of compositions inside composition constraints.

Patterns are translated differently into propositional formulas than completely defined compositions so we must replace the p function[7] with a new function *p'*. The function *p'* translates a pattern, i.e., a set of features that are defined to be selected and a set of features that are defined to be not selected, into a propositional formula ($p' : \mathcal{P}(\mathbb{F}) \times \mathcal{P}(\mathbb{F}) \to \mathbb{P}$). In a formula that p translated from a composition, every feature was defined to be selected or not selected. In a formula that *p'* translates from a pattern, only features are defined that decide whether the tree's code element exists with a particular identifier. That is, all features that are not inside the pattern remain undefined in the translated formula. The pattern that describes for *ListLlist* the compositions where List exists is $\neg \mathit{ListAdt} \wedge \mathit{Base}$ (cf. Fig. 4b) where *GetPop*, which does not decide the existence of List, is undefined. Since, this pattern is already a propositional formula it is not translated further.

To reduce the size of decision trees even more, we merge specific nodes when they expose the same identifier. As a consequence, decision trees become directed graphs when one identifier can be created with different feature selections. As we show in Section 4.3, nodes with equal identifiers cannot be merged generally.

[6] If an identifier exists in multiple trees, safe composition of preceding monotonic feature modules and RFMs guarantees the identifier does not occur twice in a particular program.

[7] Function *p* translated feature compositions into propositional formulas, cf. Sec. 4.1.

Program	Refactorings	#SLOC*	#MFM$^\diamond$	#id† from MFMs$^\diamond$	#id† from RFMs	max. TH$^\flat$	avg. TH$^\flat$
ADT library	1x rename class, 4x rename method	11	1	7	9	5	2
Workbench.texteditor	1x rename class, 2x rename field	∼16K	2	3428	68	3	1.02
Workbench.texteditor #2	27x rename class, 28x rename field	∼16K	2	3428	2538	55	1.53
GPL	2x rename class, 2x rename method, 18x encapsulate field, 2x extract interface	∼1K	15	160	252	4	1.89
ZipMe	1x rename class	∼3K	14	656	18	2	1.03
Raroscope	2x rename class	∼250	5	57	54	2	1.9

*lines of source code without RFMs; $^\diamond$monotonic feature module; †identifier; $^\flat$tree height

Table 1. Information on case studies.

Additionally, we implemented technical optimizations for decision trees. We omit their discussion because they are not important for the presented concept.

4.3 Preconditions on Inheritance Hierarchies

It is not enough for some refactorings that code elements with certain identifiers exist. Additionally to the existence of single code elements, a number of refactorings require properties of inheritance hierarchies [29, 33].[8] As an example, the result of a 'Rename Method' refactoring is well-typed but regarded as incorrect when it creates a method that overrides an existing method the transformed method did not override before the refactoring (an error called 'method capture') [29, 26, 33]. In order to verify that such errors do not occur, we must consider relationships between code elements (and thus between nodes of different decision trees).

We maintain the relationship of code elements, known already from the type system. For example, a decision tree node which represents a method references a node (in a different tree) which represents this method's host class. As a result, we can compute connections between nodes of different code elements/trees, e.g., calculate the methods a method, that is to be renamed, may capture.

As an example, when a single identifier occurs in different decision trees then *different* code elements can have the same identifier in different configurations (e.g., after refactorings apply). Although they have equal identifiers, these code elements still may differ in their relations to other code elements. For instance, consider a method $\mathtt{x}_1$, which has the identifier $\mathtt{x}$ in one configuration, is in a class that has a superclass. Consider further, a *different* method $\mathtt{x}_2$, which has the same identifier $\mathtt{x}$ in another configuration, is in a class without superclass. As a consequence, $\mathtt{x}_1$ may capture a different method but $\mathtt{x}_2$ cannot. When method $\mathtt{x}_1$ applies, the method that can be captured must not exist – we assure this by relating patterns of both methods.

Nodes with equal identifiers cannot be merged generally. To deal with the above example and similar cases, we must distinguish code elements $\mathtt{x}_1$ and $\mathtt{x}_2$ although they have equal identifiers in decision trees. We do not merge respective nodes. Two nodes can only be merged when they have equal identifiers *and* the same relations to other nodes (e.g., to the same host class node).

5. Case Studies

Previously, we implemented RFMs and a tool to compose RFMs together with monotonic feature modules [23]. To verify safe composition for RFMs, we extended Thaker's safe composition algorithm [35] with our additions described in the previous section.[9]

We now report on four case studies. First, a library of *abstract data types (ADTs)* with transformations was written independently of this work. This study was our proof of concept because it is rather small. Second, we verified safe composition for RFMs applied on a large-scale Eclipse library to verify that storing the trees does not exceed memory. Third, to verify that current computers can create and evaluate trees in reasonable time for a high number of refactorings in a large-scale program, we report on an extended study of the Eclipse library. Finally, we verify RFMs that were added to an existing feature-oriented design of a software product line, i.e., where numbers of monotonic feature modules can apply and create identifiers *before* the RFMs apply. We summarize properties of the analyzed programs in Table 1.

Abstract data type library. With the running example in this paper, we lean on a former study on abstract data types. Although the feature model of this study is moderately complex, e.g., it subdivides features, we expected this study to be easy because its implementation is small (only 11 lines of source code). We were surprised that our verifier still discovered mistakes, because the design looked correct – after we had a closer look we found the error inside our model and corrected it.

Technically, this study was small: it comprises only 7 identifiers of members and classes that are created from one monotonic feature module. Just five RFMs could be selected to create 9 additional identifiers (resulting in 16 identifiers). The maximal height of an optimized decision tree in this study was five and the average height of all trees was just 2. Our verifier created the trees and evaluated them in 0.2 seconds[10] which was obviously much faster than our manual attempt.

'Workbench.texteditor'. This study was inspired by work of Dig et al. [14]. Commonly, monotonic feature modules are large-scale building blocks that may include a large number of classes and members, i.e., a large number of identifiers to manage. To measure the size of decision trees in large-scale programs we verified a study that we performed when we discussed RFMs [23]. In

[8] 14 of the 23 covered refactoring types (cf. Sec. 2.2) have preconditions on inheritance hierarchies.

[9] We use the Sat4j SAT solver (http://www.sat4j.org/). We reuse Thaker's tool and the tools it relies on, e.g., AHEAD's bcj2j or bccompiler. In line with these tools, our prototype does not support packages yet – our theoretical approach however is not restricted this way. We moved the classes of all case studies into the default package.

[10] All measurements in this paper were performed on an Intel Centrino with 1.5GHz and 512MB RAM. They are meant as hints, e.g., they do not include checks for safe composition of included monotonic feature modules or calculating statistics.

this study, we reimplemented, with monotonic feature modules and non-monotonic RFMs, the transformations performed on a large Eclipse library between two releases of this library. The transformations were recorded by the library's revisioning system. That is, features in this study represent selectable development steps.

We moved the large Eclipse library 'workbench.texteditor' (∼16K lines of source code) into a monotonic feature module and refined it with a monotonic feature module and non-monotonic RFMs. One RFM renames a class Levenstein into Levenshtein because this change was recorded in the revisioning system, two other RFMs renamed two fields which have Levenstein as their static type. The 'Rename Field' RFMs are arranged *after* the 'Rename Class' RFM and reference the fields using the new classname Levenshtein (as class Levenstein does no longer exist). Thus, both depend on the 'Rename Class' RFM which creates Levenshtein. The verifier alerts that the feature model we used so far for this study (every monotonic feature module and every RFM is declared as independent) allows feature combinations that have errors when composed. Specifically, the verifier alerts us that both 'Rename Field' refactorings depend on the 'Rename Class' refactoring but can be selected without it. The verifier proposes to add this dependency to the model in order to make it safe. After we corrected our model all legal feature combinations compose safely.

The two monotonic feature modules in this study are large (they comprise 3428 identifiers) but the three RFMs only create 68 additional identifiers. With our optimized trees we managed these 3496 identifiers easily and did not run into memory problems. Our verifier created the trees and evaluated them in 1.6 seconds.

'Workbench.texteditor' #2. Next, we verified how the performance is affected when we apply a large number of RFMs to the (large) Eclipse library 'workbench.texteditor'. We applied 55 RFMs onto the 'workbench.texteditor' library. We renamed class Levenstein 27 times, renamed the field DefaultCellComputer.levenstein of type Levenstein 1 time, and renamed field OptimizedCellComputer.levenstein of type Levenstein 27 times. We chose these refactorings to create high decision trees with different refactorings.

This study starts with 3428 identifiers that are created from two monotonic feature modules (i.e., 3428 decision trees). Fifty-five RFMs then can create 2538 additional identifiers (resulting in 5966 identifiers) that were managed in our trees. Our verifier created the optimized trees and evaluated them in an acceptable 6.7 seconds. In this study our tree optimizations became very important: without optimizations all 3428 trees would have a height of 57 nodes (2 monotonic feature modules + 55 RFMs) and every tree would have 2^{56} leaf nodes as every tree is binary and balanced, but due to our optimization we managed just 5966 nodes altogether. Interestingly the average height of all optimized trees remains rather small (1.53 nodes).

After we were corrected by our tool in the small ADT case study, we were not really surprised when our verifier revealed errors in this study. For instance, we did not notice in our model that certain RFMs depend on each other. With the verifier, we also found multiple spelling mistakes inside our RFMs, e.g., when fields to be renamed could not exist at all. Still, we were surprised about the number of mistakes we found with our verifier.

Graph Product Line (GPL). RFMs integrate into feature-oriented design so we must test whether the approach works when numbers of monotonic feature modules can create identifiers before RFMs apply. In prior work we explored the facilities of RFMs and applied RFMs to a standard product line of graph data structures (proposed as benchmark for product line technologies) [25].

Fifteen monotonic feature modules of GPL create identifiers of members and classes. Twenty-four RFMs can be selected in this study and can create a number of identifiers as well. The GPL features are related to each other with complex composition constraints which seriously interfered with our manual verification attempt. Our verifier created the decision trees and evaluated them in 1.5 seconds while we needed minutes to confirm composition errors – several times we doubted our verifier until we found the subtle errors in the verified code (of the RFMs) or feature model.

Further studies. In prior work [23], we applied one RFM to a version of the ZipMe library[11], in which 14 monotonic feature modules (like *Crc* or *Checksum*) create identifiers, in order to integrate this library with legacy clients. Special to this study is the number of variants – namely, we could compose 2^7 different variants of ZipMe. Our verifier created and evaluated the trees for this study in 0.2 seconds without composing the large number of variants.

We also verified a study from prior work where we integrated a version of the compression library Raroscope[12] with legacy clients. In this library, 5 features (like *Crc* or *OperatingSystem*) create identifiers – two RFMs succeed these features. Our verifier created the trees of this study and evaluated them in 0.1 seconds – we did not have to compose all 2^4 Raroscope variants.

Summary. Our prototype implementation found errors which would have gone unnoticed without it. We found that our prototype implementation verified complex feature models with numbers of large-scale monotonic feature modules and non-monotonic RFMs in no time. However, overall we found that feature modules that create and remove code (like RFMs) increase complexity *rapidly*. Specifically, even small programs became hard to verify manually (several times we doubted the verifier until we found the alerted subtle errors). We believe that non-monotonic features have potential to increase expressiveness and might be suited for several use cases like refactorings, but we argue: When such powerful feature transformations (feature modules) apply we need automated concepts and tools as presented in this paper to verify resultant designs.

6. Related Work

Safe composition of transformations. Thaker et al. [35] determined composition constraints for monotonic feature modules and used the constraints to verify safe composition. Delaware et al. extended this work and proved the underlying type system to be sound [13]. Similarly, Apel et al. defined a product line type system and proved it sound [2]. They all assume that feature modules monotonically add code. Czarnecki applied safe composition techniques to transformations performed on feature-based model templates, transformations that monotonically remove elements [11]. CIDE guarantees safe composition for ifdefs (represented as colors) [19, 21]. Ifdefs delete code monotonically. In this paper we guarantee safe composition for refactorings as part of feature modules, i.e., feature transformations that create and delete code.

Whitfield et al. aims at safe composition of code transformations like 'dead code elimination' or 'loop unrolling' [36]. With found dependencies, they aim to find an ordering for transformations and to warn developers against interactions between transformations. We verify that a set of *feature transformations* (monotonic and non-monotonic) can be combined as defined in a *feature model*.

Model checking. Researchers use model checking technology to verify properties for programs, e.g., [15], or program transformations [24]. In contrast to their work, we use SAT technologies to verify *configurable sequences* of refactoring transformations that are performed on (possibly) *configurable* base code.

[11] http://sourceforge.net/projects/zipme/

[12] http://code.google.com/p/raroscope/

Graphs as program representations. Compilers usually represent programs as graphs, called abstract syntax trees [28]. Some researchers perform refactorings on enriched graphs that represent programs [17]. Others formalize refactorings as graph rewrite rules to guarantee they preserve program properties [26]. All these approaches do not verify sequences of selectable refactorings on a (configurable) base code.

Sequenced program transformations. Roberts defined preconditions for refactorings and showed how to combine preconditions of sequenced refactorings [32]. Roberts derives composite preconditions for single sequences but does not verify *configurable* sequences against a *model*.

Several approaches for meta-programming exist like [34, 6]. The transformations in these approaches are commonly non-monotonic like RFMs. A possible direction of future work is to extend our concept of decision trees to verify safe composition for such approaches as well. For that we need to figure out the preconditions for each meta-program (and implement a template test for these preconditions on decision trees). For some of them we need to record method bodies in decision trees too – this will also increase the coverage of standard refactorings from [16].

7. Conclusions

Programs can be composed by successively applying transformations that add features to a program. Program transformations must be verified to apply without errors. However, we cannot test every combination of transformations as there can be millions of combinations.

Prior work focused on safe composition of transformations that monotonically add code in order to produce program variants. We generalized prior work in that we automatically verified safe composition for transformations that add *and* remove code. Specifically, we detect errors for automated refactorings that transform a program when selected. We implemented the proposed concepts and tested them using case studies.

We found that the complexity to verify safe composition of transformations, that can add and delete code, grows *rapidly*. Specifically, we experienced subtle mistakes when we manually verified even small studies. We argue that automated analyses as presented in this paper are essential to apply and manage more powerful transformations than those already known.

Acknowledgments

Martin Kuhlemann was supported and partially funded by the *DAAD Doktorandenstipendium* #D/07/45661. Batory's work was supported by NSF's Science of Design Project #CCF-0724979. The authors thank Norbert Siegmund and Maider Azanza for helpful discussions and hints on earlier versions of this paper.

References

[1] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[2] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type-safe feature-oriented product lines. Technical Report MIP-0909, Department of Informatics and Mathematics, University of Passau, 2009.

[3] S. Apel, M. Kuhlemann, and T. Leich. Generic feature modules: Two-staged program customization. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, pages 127–132, 2006.

[4] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[5] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 36–50, 2008.

[6] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 265–279, 2005.

[7] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20, 2005.

[8] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[9] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[10] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 191–199, 1993.

[11] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220, 2006.

[12] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 23–34, 2007.

[13] B. Delaware, W. Cook, and D. Batory. A machine-checked model of safe composition. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35, 2009.

[14] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 441–450, 2008.

[15] D. R. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 191–210, 2004.

[16] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] N. Juillerat and B. Hirsbrunner. Food: An intermediate model for automated refactoring. In *International Conference on Software Methodologies, Tools and Techniques (SoMeT)*, pages 452–461, 2006.

[18] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[19] C. Kästner and S. Apel. Type-checking software product lines - A formal approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 258–267, 2008.

[20] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320, 2008.

[21] C.H.P. Kim, C. Kästner, and D. Batory. On the modularity of feature interactions. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–34, 2008.

[22] C. W. Krueger. New methods in software product line practice. *Communications of the ACM (CACM)*, 49(12):37–40, 2006.

[23] M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. In *Proceedings of the International Conference on Software Reuse (ICSR)*, 2009.

[24] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 283–294, 2002.

[25] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24, 2001.

[26] T. Mens, N. v. Eetvelde, D. Janssens, and S. Demeyer. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[27] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Conference on Design Automation (DAC)*, pages 530–535, 2001.

[28] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[29] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[30] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering (TSE)*, SE-2(1):1–9, 1976.

[31] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443, 1997.

[32] D. B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[33] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 268–285, 1996.

[34] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Workshop on Reflection and Software Engineering*, pages 117–133, 2000.

[35] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104, 2007.

[36] D. L. Whitfield and M.L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, 1997.