Technical Report

Nr.: FIN-007-2009

Safe Composition of Refactoring Feature Modules

Martin Kuhlemann, Don Batory, Christian Kästner

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-007-2009

Safe Composition of Refactoring Feature Modules

Martin Kuhlemann, Don Batory, Christian Kästner

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG):

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Martin Kuhlemann
Postfach 4120
39016 Magdeburg
E-Mail: martin.kuhlemann@ovgu.de

http://www.cs.uni-magdeburg.de/Preprints.html

Auflage: 52

Redaktionsschluss: 23.04.2009

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

Safe Composition of Refactoring Feature Modules

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

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

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

Abstract

Programs can be composed by successively applying transformations that add features to a code base. These transformations must apply without errors but we cannot test every combination of them. We must detect errors automatically to encapsulate single transformations and scale them. Prior work focused on transformations that monotonically add code in order to produce program variants. We generalized their work in that we automatically detect composition errors for transformations that add *and* remove code. Specifically, we detect errors for automated refactorings that transform a program when selected. The generalization is important to detect errors for modules that add and remove code. As a result, we can now guarantee that refactorings and sequences of refactorings compose without errors in feature-oriented designs.

1. Introduction

Programs in software product lines are created by composing features, which are code transformations that encapsulate an increment in program functionality [17, 4]. Different combinations of features produce different programs; reuse is inherent as many programs share the same features.

Not all combinations of features are meaningful [3]. Legal combinations are defined by a feature model which declares features to be mandatory, optional, alternative or inclusive to other features [9]. Unfortunately, developers cannot verify properties of all programs in a product line by brute force as the number of programs can be up to millions [13]. One solution to this problem is *safe composition*, a technique that has been used to prove that all programs of a product line that are assembled from feature modules are type correct [23, 12, 10]. In prior work, feature modules were monotonic transformations 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.

We recently proposed that feature modules be extended in a fundamental way: to include object oriented refactorings [16], which can rename, add, and delete existing classes and members. Feature modules are now more expressive and break with the traditional assumption of monotonicity [1] where only code artifacts can be added. A feature can now rename a method M to method N. If a subsequently added feature references M, the resulting program is no longer type safe and will not compile. Prior safe composition analyses could not detect this error. In this paper, we show how safe composition analyses can be generalized to allow features that include non-monotonic transformations.

(a) diagram notation (b) formula notation

Figure 1. Sample feature model.

2. Background

2.1 Safe Composition in Feature-Oriented Design

Feature models. A *feature* is an increment in program functionality [4]. 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 by a feature model [9]. A feature model can additionally define the order in which selected feature modules are composed [3].

A feature model is depicted by a feature diagram (see Figure 1a). This model defines an abstract data type List with the features *Base* and *Lock* and maps these features to the feature modules *Base* and *Lock* respectively. The model says that any combination of *Base* and *Lock* yields a meaningful program, i.e., the legal compositions according to this feature model are *Base*, *Lock*, and *Lock*•*Base* (*Lock* applies transformations to *Base*). We graphically encode the feature order by reading from left to right 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 in Figure 1b as propositional formula following the rules of [23, 2, 6]. Each variable in this

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.

formula corresponds to one feature – we thus call it 'feature variable'. A feature variable is 'true' when the according feature is selected and 'false' otherwise. The formula is true for legal feature selections.

Feature modules. Features are implemented by feature modules which successively transform the code contributions of prior features [19, 4]. 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 1a (we use the Jak language [4] 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 calls the refined method using Jak's keyword Super (Line 14) and adds statements.

Safe composition of refinements. The feature model in Figure 1 does not represent the set of programs which can be composed from the referenced feature modules of Figure 2. Any composition of *Base* and *Lock* is legal for this model. However, *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 the legal 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. [23] determined dependencies between feature modules, called *composition constraints*, 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.

2.2 Refactoring Feature Modules

There are many use cases where more expressive feature modules are needed, i.e., feature modules that can create and delete code. An example is adding refactorings to feature modules.

A *refactoring* transforms a program by altering the program's structure but not its semantics [8]. For example, modifying a method's name and updating its references is a 'Rename Method' refactoring. The standard refactorings that we use to illustrate the concepts in this paper are 'Rename Method' and 'Rename Class' but our approach is not limited to them.[1]

In previous work, we explained how refactorings could be included in feature modules; we called such a module a *refactoring feature module (RFM)* [14]. Figure 3 shows RFMs in the order they can be applied to the feature module *Base* (top-down order). In line with feature modules, 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 into ADT of feature modules *Base* and *GetPop* when they are selected but not List of *ListLlist* (*ListLlist* is arranged after *ListAdt*).

As an aside, RFMs are beneficial for product lines of components [14]: A component generated from feature modules often needs to integrate with legacy applications [5], which expect particular names and signatures for a component's interface, that is, 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.

In this paper we focus on RFMs that modify fully-qualified names of code artifacts ('identifiers' for short). Thus, refactorings which affect only method bodies (e.g., 'Remove Assignments to Parameters'[2]) are not considered. The standard refactorings that we cover are: Add parameter, Change Unidirectional Association to Bidirectional, Change Value to Reference, Encapsulate Field, Extract Class, Extract Interface, Extract Superclass, Hide Delegate, Inline Method, Introduce Parameter Object, Move Class, Move Field, Move Method, Rename Class, Rename Field, Rename Method, Replace Constructor with Factory Method, Replace Method with Method Object, Self Encapsulate Field. That is, we cover about 26% of the standard refactorings in [8]. This list contains the most common refactorings. According to our experience and prior studies this is no restriction.

[1] 'Rename Class' changes the name of a class [8].

[2] 'Remove Assignments to Parameters' assigns method parameters to new local variables and uses these variables inside the method instead [8].

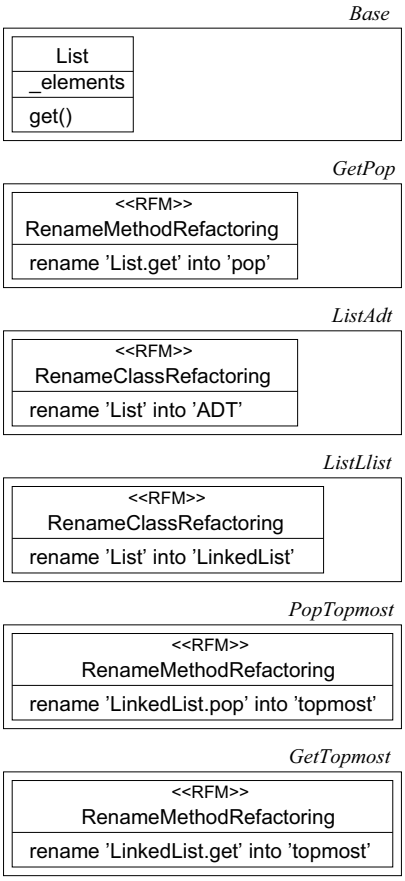

Figure 3. Refactorings as parts of feature modules.

Given the above, we want to verify that all legal combinations of feature modules (including RFMs) that are permitted by a feature model are type safe. We examine the difficulty of doing so in the following sections.

3. Analysis of Refactorings

The key to the safe composition of a refactoring is to verify its *preconditions* [20, 8, 16]. Preconditions are formulated in terms of code artifacts (described with identifiers) which must exist when the according RFM applies and artifacts which must not exist. For instance, a refactoring which renames class List into LinkedList requires both List to exist and LinkedList to not exist. If List does not exist then the refactoring fails because there is no class to rename. If LinkedList exists in the code to refactor then the refactoring fails too because there can only be one LinkedList class in a program [16, 20]. 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 for the refactorings of Figure 3:

- The 'Rename Method' refactoring in RFM *GetPop* requires a method List.get. *Base* creates List.get and thus *GetPop* must always appear after *Base*. *GetPop* additionally requires that List.pop does not exist – this is always true because List.pop is not created by any combination of features prior to *GetPop*. The composition constraint for *GetPop* is that when *GetPop* applies then *Base* must have been applied before (*GetPop*→*Base*).
- The 'Rename Class' refactoring in RFM *ListAdt* requires a class List and that a class ADT does not exist. List is created by *Base* and ADT is not created before *ListAdt* thus the composition constraint for *ListAdt* is *ListAdt*→*Base*.
- The 'Rename Class' refactoring in RFM *ListLlist* requires a class List and that a class LinkedList does not exist. Special to *ListLlist* is *ListAdt*. *ListAdt* deletes List and so *ListLlist* requires that *ListAdt* not be present. The constraint for *ListLlist* is *ListLlist*→(¬*ListAdt*∧*Base*).
- The 'Rename Method' refactoring in RFM *PopTopmost* requires LinkedList.pop. *ListLlist* creates LinkedList.pop when *GetPop* is selected (*ListLlist* creates LinkedList.get without error when *GetPop* is not selected). Thus, *PopTopmost* requires both *ListLlist* and *GetPop*.[3] The constraint becomes *PopTopmost*→(*ListLlist*∧*GetPop*).[4]
- Special to *GetTopmost* is that it requires method LinkedList.get but this method is only created by *ListLlist* when *GetPop* is *not* selected (*ListLlist* creates LinkedList.pop without error when *GetPop* is selected): *GetTopmost*→(*ListLlist*∧¬*GetPop*).

Summary. In safe composition of feature modules, a feature F requires another feature G if F references an element (e.g., method, field, class, interface) introduced in G. In safe composition of RFMs, one RFM may require that multiple features apply in an ordered composition to create an artifact with a particular identifier. Finally, RFMs do not only require that certain features are selected but also that certain features are not selected.

4. Safe Composition of RFMs

In this section, we present our solution to validate safe composition of RFMs without composing and testing every legal composition. We explain the overall process of our approach in Section 4.1 and its implementation in Sections 4.2 and 4.3.

4.1 Basic Concept

Preconditions of refactorings reference identifiers of artifacts that either must exist or that must not exist. We determine composition constraints by relating feature compositions to each other that make the referenced artifacts exist (or not exist). We translate these constraints to propositional

[3] *Base* creates List.get; *GetPop* renames List.get into List.pop; *ListLlist* renames List.pop into LinkedList.pop

[4] Note, *PopTopmost* also requires that *ListAdt* does not apply and *Base* applies but this is guaranteed transitively by the constraints of required *ListLlist* and *GetPop*.

formulas as SAT solvers can validate them efficiently in one step for all feature combinations.

To provide a concise syntax for the next discussions, we explain how feature compositions are translated into propositional formulas: A composition of features translates into a conjunction of feature variables[5] 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, we translate a feature composition *ListAdt*•Base that does not select *GetPop* into the formula *ListAdt*∧¬*GetPop*∧Base.

When we examined preconditions of refactorings we observed that they fall into two camps: the existence of some artifacts $\mathbb{X}$ and the non-existence of some artifacts $\mathbb{Y}$ in the code to refactor. As a consequence, two preconditions emerge where elements of $\mathbb{X}$ must be guaranteed to exist and elements of $\mathbb{Y}$ must be guaranteed to not exist. 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 elements of $\mathbb{X}$ exist and elements of $\mathbb{Y}$ not-exist immediately prior to that refactoring.

As an example, suppose a refactoring *R* requires some artifact with the identifier x$\in \mathbb{X}$ then we validate (a) that x always exists immediately prior to *R*. Further, suppose that a refactoring *R* requires some artifact with the identifier y$\in \mathbb{Y}$ to not exist then we validate (b) that y never exists immediately prior to *R*.

(a) With an assumed function (its implementation is not important for now) we calculate the set $\mathbb{C}$ of unique feature compositions that make an artifact with the required identifier x exist at the point immediately before the feature *R*.

$$\mathbb{C} = \{C_1, C_2, \ldots C_n\} \tag{1}$$

Consequently, a constraint which we must validate for all legal compositions is that *R* implies that one composition of $\mathbb{C}$ applies prior to it:

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

Technically, SAT solvers efficiently validate existential clauses for propositional formulas [15]. Therefore, we transform constraint (2) into a theorem to verify by a SAT solver. That is, instead of validating that every legal composition fulfills *R*'s composition constraint we validate whether there is a composition that is legal for a feature model but that does not fulfill *R*'s constraint. We follow previous work [2, 6, 23] to translate a feature model into a propositional formula FM. The theorem that we need to prove is:

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

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

This formula (3) is unsatisfiable when *R* composes safely. If (3) is satisfiable then the binding tells us a legal composition of features for which *R*'s precondition fails (because x does not exist).

(b) We now consider how to verify the non-existence of an artifact. With our assumed function we calculate the set $\mathbb{C}'$ of feature compositions that make y exist at the point immediately before *R* where *R* requires y to not exist.

$$\mathbb{C}' = \{C'_1, C'_2, \ldots C'_n\} \tag{4}$$

Consequently, a constraint which we must validate for all legal feature compositions is whether *R* implies that no composition of $\mathbb{C}'$ applies prior to it:

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

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

When the proofs of either (a) or (b) fail then we found an error in the feature model, the feature modules, or both. That is, the feature *R* is selected but the precondition of *R* is violated. A domain expert is alerted and can now correct the feature-oriented design. If finally both tests succeed (the SAT-tests fail) then no legal feature composition causes errors when composed – all legal programs can now be composed safely.

Example. We calculate the composition constraint for *ListLlist* of Figure 3 systematically which we determined informally in Section 3. *ListLlist* renames class List into LinkedList. Thus, *ListLlist* requires (a) that List exists and requires (b) that LinkedList does not exist in all legal compositions immediately before *ListLlist*.

(a) To validate the existence of List, we calculate the set of compositions which make List exist immediately before *ListLlist* ($\mathbb{C}$={*Base*,*GetPop*•*Base*}). From this set we calculate the composition constraint for *ListLlist* and translate this constraint into a propositional formula:

$$\begin{aligned} ListLlist \rightarrow ((&\neg ListAdt \wedge \neg GetPop \wedge Base) \\ &\vee(\neg ListAdt \wedge GetPop \wedge Base)) \end{aligned}$$

This formula was directly translated from $\mathbb{C}$ and for clarity it is not optimized – we show optimizations later.

Instead of validating that every legal composition fulfills *ListLlist*'s constraint we validate an equivalent theorem, i.e., whether there is one composition that is legal for a feature model FM but that does not fulfill *ListLlist*'s constraint.

$$\begin{aligned} FM \rightarrow \neg(ListLlist \rightarrow ((&\neg ListAdt \wedge \neg GetPop \wedge Base) \\ &\vee(\neg ListAdt \wedge GetPop \wedge Base))) \end{aligned}$$

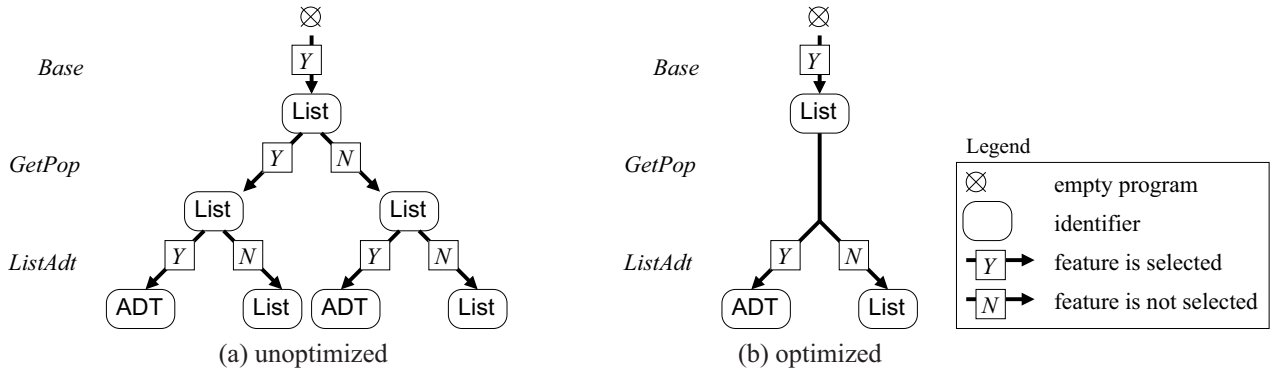

Figure 4. Transformation history for List of Fig. 3.

If the SAT solver finds an assignment for the variables in this formula this assignment corresponds to a feature composition which is in error, i.e., a composition which is legal, contains *ListLlist*, and does not fulfill the constraint of *ListLlist*. These compositions do not contain the required class List.

(b) To validate the non-existence of LinkedList, we calculate the set $\mathbb{C}'$ of compositions that create LinkedList immediately before *ListLlist* ($\mathbb{C}'=\varnothing$). From this set we create a composition constraint and then a propositional formula. We finally validate this formula with a SAT solver as shown above. Since $\mathbb{C}'$ here is empty the constraint becomes a tautology and no composition can violate it (the SAT-test fails).

4.2 Computing Feature Compositions with Identifiers

In order to assemble composition constraints we relied on a function which determines compositions where an artifact with a particular identifier exists. In this section, we show (with optimizations) how we calculate those compositions without actually enumering them.

We record all identifiers of artifacts that feature modules create and establish each identifier as root of a decision tree – when 2 features introduce the same artifact then 2 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. Refinements do not transform identifiers of artifacts and so features that solely encapsulate refinements are not recorded at all. A single decision tree represents one code artifact that has different identifiers over time (after refactorings apply). A tree represents the history of decisions on features that lead the identifier of its root node to identifiers of leafs – we thus call it *history tree*.

In Figure 4a we visualize the effects of feature decisions prior to *ListLlist* (*Base*, *GetPop*, and *ListAdt*) from Figure 3 on the identifier of class List, i.e., we show the history tree of List. In this tree, an arrow is a feature which can transform an artifact'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.

To create history trees we iterate the features in the order that is defined in the feature model. In each iteration step we first validate 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. Trees we use to validate an RFM thus contain only effects of features that precede the currently validated RFM (as RFMs only transform code added prior to them).

With history trees for all identifiers we can determine compositions which include artifacts with a particular identifier. When an RFM requires that an artifact of a particular identifier exists immediately before the RFM then it requires a *leaf node* with this identifier in the history trees which are recorded up to this RFM. The composition that makes the tree's artifact with its leaf's identifier exist before the validated RFM emerges from the path of feature decisions from this leaf backward to the empty program.

To exemplify the use of history trees we recalculate the compositions prior to *ListLlist* that include List (in the prior section we assumed a function 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 history tree we calculate two different compositions (paths to the empty program) that contain List so $\mathbb{C}=\{Base, GetPop \bullet Base\}$. We translate and use these compositions in order to create composition constraints and validate these constraints with SAT technologies.

Optimization. The history tree of every code artifact grows exponentially with the number of features it records. To optimize history trees, we do not add children to a node when both children have the same identifier as the current node, i.e., when the refactoring does not change the identifier of a code artifact. 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 history tree for List is shown in Figure 4b.

With optimized history trees we compute *patterns* for compositions with an artifact of a particular identifier when we calculate paths from leafs to the empty program. That is, we do no longer calculate completely defined compositions. Inside such a pattern only features appear which decide whether an artifact 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.

In contrast to completely defined compositions, patterns have a three-value logic because patterns distinguish (a) features that must be selected in order to create the tree's artifact with the required identifier, (b) features that must not be selected, and (c) features that do not influence the creation of the tree's artifact with the required identifier. Therefore, patterns are translated differently into propositional formulas than completely defined compositions. In a formula translated from a composition, every feature is defined to be selected or not selected. In a pattern only features are defined that decide whether the tree's artifact exist with a particular identifier and only those features contribute to the translated formula. 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 ¬ListAdt∧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 – we use these patterns instead of compositions inside composition constraints.

To further optimize the tree concept we merge nodes *of one tree* when they expose the same identifier. As a consequence, history trees become directed graphs when one identifier can be created with different features from one root. A path description (i.e., a pattern) to the empty program is then more complex because it combines alternative paths but identifiers are unique in leafs of one tree. As we show in Section 4.3, leafs with equal identifiers *in different trees* cannot be merged (different trees cannot be intermingled).

We implemented several technical optimizations for history trees. We omit their discussion for clarity and brevity of this paper.

4.3 Preconditions Towards Inheritance Hierarchies

It is not enough for some refactorings that artifacts with certain identifiers exist, i.e., the composition result would be type-safe but incorrect. Additionally to the existence of single artifacts, a number of refactorings require properties of whole inheritance hierarchies [16, 22].[6] As an example, a 'Rename Method' refactoring is incorrect when the method it creates overrides a method the transformed method did not override before the refactoring (so-called 'method capture' [22]). In order to validate such errors do not occur, we must navigate between artifacts and so between nodes of different history trees.

In order to navigate between nodes of different history trees we associate them according to the type of artifact they represent (methods, fields, or classes). A node that represents a method references a node which represents this method's host class (in a different tree). As a result, we can navigate between nodes of different artifacts/trees and calculate the methods a method, that is to be renamed, may capture.

Different history trees cannot be intermingled. When a single identifier occurs in different history trees then different code artifacts can be transformed to expose it. These artifacts may differ in the inheritance hierarchy they are in or in the position inside an inheritance hierarchy. For example, a method which has the identifier x in one configuration is in an inheritance hierarchy but a different method which has the identifier x in another configuration is not. We thus must distinguish different artifacts with equal identifiers in our history trees – we cannot merge identifiers to intermingle history trees.

5. Related Work

Safe composition. Thaker et al. [23] determined composition constraints for feature modules and used the constraints for safe composition. Delaware et al. extended this work and defined a sound type system of composition constraints [7]. Both assume, feature modules monotonically add code. CIDE guarantees safe composition for ifdefs (represented as colors) [11], a mechanism formalized by Kim et al. [12]. Ifdefs delete code monotonically. In this paper we guarantee safe composition for refactorings as part of feature modules, i.e., modules that create and delete code.

Whitfield et al. guarantees safe composition for code transformations [24]. The transformations of Whitfield are optimizations as used in compilers, like 'dead code elimination' or 'loop unrolling', but they are no object-oriented refactorings. Whitfield relates types of transformations and proposes a composition order for transformations of these types, e.g., 'loop unrolling' enables 'dead code elimination' and thus should be arranged before. The transformations of Whitfield do not consider type-safety as their transformations are on statement level. A possible direction of future work is to combine both approaches.

Model checking. Sittampalam et al. used model checking to validate properties of incrementally executed transformation specifications [21]. The transformations of Sittampalam are no object-oriented refactorings but optimizations inside methods, e.g., propagation of variable values or dead assignment eliminitation. The transformations of Sittampalam are defined completely in their transformation modules with regular expressions to match code to transform. RFMs cannot enumerate all identifiers of artifacts which the refactoring

[6] 13 of the 19 covered refactoring types (cf. Sec. 2.2) have preconditions towards inheritance hierarchies.

creates and deletes *in every feature composition* nor can give a pattern for them. An RFM also cannot enumerate identifiers of artifacts that it requires to not exist because there may be an infinite number of them. Sittampalam evaluates a sequence of transformations at once but not multiple possible sequences. Technically, Sittampalam uses Prolog to validate transformations and we use SAT technologies.

Category theory. A program corresponds to a category in category theory [18] and identifiers of program artifacts correspond to points in that category. Refactorings correspond to arrows that map points (identifiers) of one category (program) to points of another category. Our trees that record refactoring effects with associated nodes merge different categories in that transformation arrows map a tree node to another in a different category (in a different program) but some nodes belong to multiple categories (programs), e.g., the refactoring 'Rename method LinkedList.get into topmost' maps a node LinkedList.get of one category to a tree node LinkedList.topmost of another category – however, both method nodes reference the same LinkedList node to represent their hosting class – this class node contributes to both categories. History trees thus visualize categories of identifiers and arrows of identifier transformation. In terms of category theory, history trees become *commuting* graphs when different paths exist to one identifier.

6. Conclusions

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

We must detect errors automatically to encapsulate single transformations and scale them. That is, users must be able to rely on the correctness of single transformations without inspecting them and every possible peace of code they may be applied to.

Prior work focused on safe composition of transformations that either monotonically add or monotonically remove code in order to produce program variants. We generalized their work in that we automatically detect composition errors for transformations that add *and* remove code. Specifically, we detect errors for automated refactorings that transform a program when selected.

The contribution of this work is important to detect errors for modules that add and remove code. Specifically, we can now validate automatically whether refactorings and sequences of refactorings compose without errors in feature-oriented designs. As a result, we can deliver configurable components, that can be configured to have a certain functionality and to have a certain interface, and guarantee that every configuration of functionality and component interface is type-safe and correct.

References

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, pages 258–267, 2008.

[12] 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.

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

[14] M. Kuhlemann, Don Batory, and Sven Apel. Refactoring Feature Modules. Technical Report 15, School of Computer Science, University of Magdeburg, 2008.

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

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

[17] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering (TSE)*, SE-2(1):1–9, 1976.

[18] B. C. Pierce. *Basic category theory for computer scientists*. MIT Press, 1991.

[19] 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.

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

[21] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. *ACM SIGPLAN Notices*, 39(1):26–38, 2004.

[22] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. *ACM SIGPLAN Notices*, 31(10):268–285, 1996.

[23] 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.

[24] 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.