Bachelor thesis
Computing Science

Radboud University

Coupled transformation of feature models and domain models for software product lines

Author:
Steven Maarse
s1010311

First supervisor/assessor:
Dr. Daniel Strüber
`dstrueber@cs.ru.nl`

Second assessor:
Dr. Cynthia Kop
`c.kop@cs.ru.nl`

March 26, 2021

Abstract

This thesis project is on marrying two research domains: graph transformations and software product lines. Graph transformations can already be applied to software product lines. However, an algorithm that executes coupled changes of the domain model and feature model by means of graph transformations is missing. In this paper, we will discuss what changes will be made to an existing algorithm, called Lifting, and how these will be expressed. The result will be a correct algorithm that solves the shortcomings of the Lifting algorithm.

Contents

Chapter 1

Introduction

In this thesis, we will be discussing concepts at the intersection of graph transformations and software product lines and extend this area.

Software product lines (SPLs) are collections of software assets combined to allow many different products to be generated based on selected features. Since within software development often very similar products are desired and they share many common features, a software product line is very efficient as it removes the need to re-implement features and to build the desired product by hand for every customer.

Developing software product lines involves maintaining a representation of the product line's features (typically, using a feature model), and making explicit how the features are related to software assets (e.g., domain models and code) [4]. For example, one type of domain model used to specify the behavior of a software system are state machines. In a state machine, many states of the system and possible transitions between these states are defined, specifying the system's behavior. As not all desired products share the same features, many states and transitions will only be present in some of the products, depending on the features associated with the product. To make explicit how features are related to states and transitions, one can annotate these elements with feature expressions, leading to a feature-annotated state machine.

Many systems, including software systems, can be modeled as graphs. Their systematic modification can then be described using graph transformations. Supported by a rich theory, graph transformations can be used to support advanced software analysis, including conflict-and-dependency analysis and model checking [1]. A graph transformation rule consists of various parts, such as the Left Hand Side, Right Hand Side and Negative Application Conditions are used to allow transformation rules to make any changes to the system. The LHS and RHS together express a “find and replace” pattern in the input graph, whereas NACs allow to further restrict this pattern. We will use transformation rules to apply changes to a software system that is modeled as a graph. For example, we can consider a state machine as a graph that models the behavior of a software system. A transformation rule could then be used to describe and apply a particular refactoring of a state machine.

Transformation rules have some restrictions on the places of the input graph they can be applied to. A place in the input graph at which we can apply the given rule is called a match. As part of our background (Sect. 2), we will revisit an existing definition of a rule application, show any conditions that need to be satisfied to proceed, explain why this is necessary, and explain how the application is performed.

Since the domain model of an SPL can be represented as a graph (with feature annotations), we can apply transformation rules to them. We will discuss how these transformation rules work and explain how an existing algorithm called “Lifting” can apply these transformation rules to an SPL. The Lifting algorithm is extremely important for applying transformation rules to a large SPL that has many features, due to the combinatorial explosion arising from the possibility to switch features on or off independently. In the preliminaries, we will explain exactly how the Lifting algorithm works, what an application of lifting looks like and why it is correct.

However, the Lifting algorithm is restricted in the type of changes it supports for modifying a software product line. Specifically, it does not allow to change the feature model by adding and removing features and constraints. It cannot add a new feature to the domain and feature model, nor can it remove an existing feature from the domain and feature model. This presents an obstacle prohibiting a broader application to realistic transformation scenarios. Therefore, we propose an algorithm that is designed to perform coupled transformations of feature models and domain models. Furthermore, we will discuss its correctness and complexity. This new algorithm will extend the existing Lifting algorithm. We will discuss the complexity and correctness of this new algorithm. Finally, a Python implementation of the algorithm is presented.

Chapter 2

Preliminaries

This chapter revisits the necessary preliminaries for this work. First, we introduce software product lines and their structure. Then, what transformation rules are and how these combine with software product lines. Finally, we show and explain "lifting" which is an essential algorithm for transforming software product lines, and the foundation of this thesis's research.

2.1 Software product lines

Developing software product lines involves maintaining a representation of the product line's features (typically, using a feature model), and making explicit how the features are related to software assets (e.g., domain models and code) [4]. For example, one type of domain model used to specify the behavior of a software system are state machines. In a state machine, many states of the system and possible transitions between these states are defined, specifying the system's behavior. As not all desired products share the same features, many states and transitions will only be present in some of the products, depending on the features associated with the product. More specifically, these states and transitions have a presence condition (PC). A PC is an expression in propositional logic over the product line's features, specifying the condition under which the annotated element is present. When a product is created, these can be evaluated, and elements whose PC evaluates to "false" are removed.

Definition 2.1.1 (Product line)**.** *A product line P consists of the following parts:*

(1) A feature model *that consists of a set of features and a propositional formula Φ_P defined over these features to specify the relationships between them.*
(2) A domain model *consisting of a set of model elements.*

(3) A mapping *from the feature model to the domain model consisting of pairs $\langle E, \phi_E \rangle$ mapping a domain model element E to a propositional formula ϕ_E over features. The formula ϕ_E is referred to as the* presence condition *of the element E.*

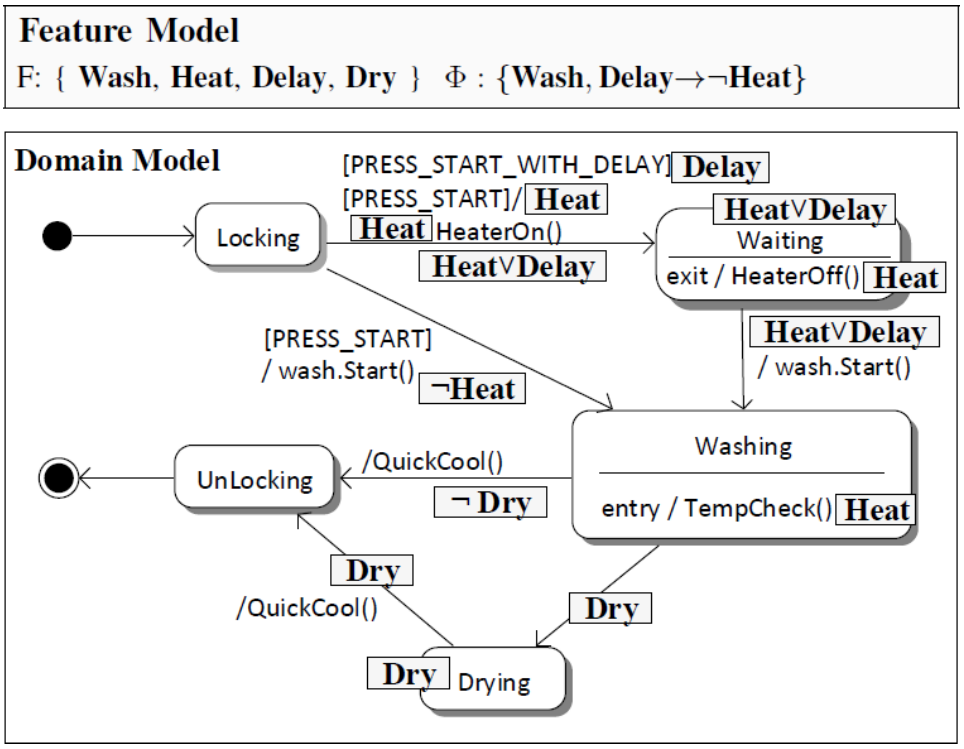


Figure 2.1: Washing Machine Controller Product Line: feature model and domain model

Consider the following example. Figure 2.1 represents the SPL of a washing machine controller. In the domain model, the rounded boxes (nodes) represent states and the edges between these boxes represent the state transitions. To capture the mapping of elements to products, some nodes and edges, like the node “Drying” have a small rectangle with bold text that shows the presence condition of the element. In this case, only products that have the feature “Dry” will have the Drying state with the edges from Washing to Drying and from Drying to UnLocking. In the feature model at the top, a set of features F and feature constraints Φ are specified as well. Feature constraints restrict the set of possible products; for example, the constraint `delay -> ¬ heat` implies that no product exists that has both features `delay` and `heat`.

2.2 Transformation rules

The second relevant area of background for this thesis are graph transformations. In graph transformations, one systematically describes changes to a (graph-based representation of a) system by encoding them into graph transformation rules ([2], explained shortly). Capturing changes as rules enables formal reasoning about them. For example, one can use a static analysis to automatically detect conflicts and dependencies between several activities expressed as rules. [8]

To modify the behaviour of a software system, these graph transformation rules can be applied to the system. A transformation rule specifies how a part of the system changes after application. Using a Left Hand Side (LHS) it specifies what part of the graph is matched to the rule, based on various constraints (for example, element types and attribute values). The Right Hand Side (RHS) then shows which nodes and edges are are added, preserved or removed.

To further restrict the circumstances under which a rule application is possible, rules can be extended with application conditions. One example use case for this is to avoid creating duplicate edges. In this case, negative application conditions (NACs) can be used to check when a LHS is not a valid matching site for the rule. If a NAC is successfully matched against the matching site, the rule cannot be applied.

Since a software product line can be modeled as a graph with states and state transitions as nodes and edges, one can apply transformation rules to SPLs to modify the behavior of products from a software product line.

Definition 2.2.1 (Transformation rule)**.** *A* transformation rule *R is a tuple $R = \langle \{NAC\}, LHS, RHS \rangle$, where LHS and RHS are the typed graphs called the* left-hand *and the* right-hand *sides of the rule, respectively, and $\{NAC\}$ represents a (potentially empty) set of typed graphs called the* negative application conditions.

Two example rules have been defined as seen in Figure 2.2. In rule 1, the incoming actions on a state x are folded into the state as an entry action. If two incoming edges of a state x have an action defined, they are removed from the edges and x will have the entry action a. The NAC specifies that x cannot already have an entry action for the rule to be applicable.

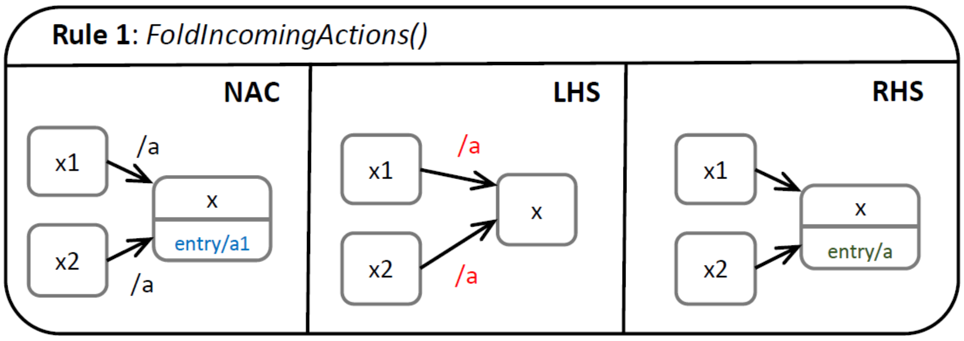


Figure 2.2: A transformation rule that folds incoming actions

2.3 Rule application

$$\begin{tikzcd} L \arrow[d, "m"] \arrow[dr, phantom, "(1)"] & I \arrow[l, "le"'] \arrow[r, "ri"] \arrow[d, "d"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m'"] \\ G & D \arrow[l, "g"] \arrow[r, "h"'] & H \end{tikzcd}$$

Figure 2.3: Rule application by a double pushout (DPO)

Definition 2.3.1 (Rule application)**.** *Let a rule $p = \langle \{NAC\}, L, R\rangle$ and a graph G with a total morphism $m : L \rightarrow G$ be given. A* rule application *from G to a graph H, written $G \Rightarrow_{r,m} H$, is given by the diagram in* Figure 2.3 *where (1) and (2) are pushouts and I is the intersection of L and R. We refer to G, m and H as a* start graph*, a* match*, and a* result graph*, respectively.*

The rule application in Figure 2.3 has L as the the LHS and R as the RHS. Here, I is the overlap between L and R. These are all of the preserved elements during rule application.

A graph rule is applied along a match m of its left-hand side to a given graph G. The application of a graph rule consists of two steps: First, all graph elements in $m(L - l(I))$ are deleted. Nodes to be deleted may have adjacent edges which have not been matched, so the rule application may produce dangling edges. Therefore, all matches m have to satisfy the *gluing condition*: If a node $n \in m(L)$ is to be deleted by the rule application, it has to delete all adjacent edges as well. If the gluing condition does not apply, the rule cannot be applied to this particular graph. Afterwards, unique copies of $R - r(I)$ are added. This behavior can be characterized by a formal construction knows as a double-pushout [5]. A pushout is a construction that abstractly captures the idea of merging two objects based on a common subobject. In case of rule applications, the sub-diagrams (1) and (2) of Figure 2.3 are both pushouts, together forming a double-pushout. Given a rule and a match, the resulting rule application is unique [5].

2.4 Lifting

A graph transformation rule can easily be applied to a single product, but when there are many possible features that a product can have, it becomes infeasible to apply a transformation rule to all products of an SPL. This is for the effort required to enumerate all possible products, which grows exponentially with the number of features. Lifting [11] is an algorithm designed to apply a graph transformation rule on an SPL to modify it such that the products derived from the SPL will be equal to the products resulting from applying the transformation rule to each product individually. The Lifting algorithm relies on SAT solving, an NP-complete problem for which existing efficient solvers are available, scaling to millions of variables (features) [6]. If a formula is satisfiable, we can say that the formula is SAT. SAT solving means to compute the satisfiability of a formula.

The Lifting algorithm in Figure 2.4 assumes the rule has already been used to find matching sites, which has brought forward a single matching site in the input. The matching site can be any part of the domain model that fulfills the constraints from the LHS; presence conditions are not yet considered at this point. At line 3, for every matching site the algorithm starts by checking if $\Phi_P \wedge \Phi_{apply}$ is SAT. A negative answer allows for skipping the transformation if there is not a single product for which all required elements on the LHS are present and for which none of the NACs apply within the SPL. If at least one NAC applies for a product, the product derived from the SPL will remain unchanged. Every element that is added according to the matching site, will be added to the domain model in lines 4 - 5 and the PC will be set to Φ_{apply} at line 6 to only affect the products the rule has matched on. Next, starting at line 8, in the for-loop, all elements that are removed will be removed just by changing the PC to include that Φ_{apply} must also not be satisfied for it to be present in the product at line 9. Only if no products remain with the element present, checked using SAT at line 10, it will be completely removed from the domain model at line 11.

```
Data: Product line $P$ with constraint $\Phi_P$, rule $R$, matching site
      $K = \langle \bar{N}, C, D \rangle$ in the domain model of P
Result: Transformed product line $P'$
$P' = P$
$\Phi_{apply} = \neg \bigvee \{\phi_N^{and} | N \in \bar{N}\} \wedge \phi_C^{and} \wedge \phi_D^{and}$
if $\Phi_P \wedge \Phi_{apply}$ is SAT then
    for $a \in A^r$ do
        add $a$ to domain model of $P'$
        $\phi'_a = \Phi_{apply}$
    end
    for $d \in D$ do
        $\phi'_d = \phi_d \wedge \neg\Phi_{apply}$
        if $\Phi_P \wedge \phi'_d$ is not SAT then
            remove $d$ from domain model of $P'$
        end
    end
end
return $P'$
```

Figure 2.4: Algorithm to apply a lifted graph transformation rule.

The rule defined in Figure 2.2 can all be applied to a software product line using the Lifting algorithm. For example, by lifting Rule 1 applying it to the SPL as defined in Figure 2.1 so that the incoming actions on the edges to the *UnLocking* state are folded into the state itself as an entry action. This results in a domain model seen in Figure 2.5.

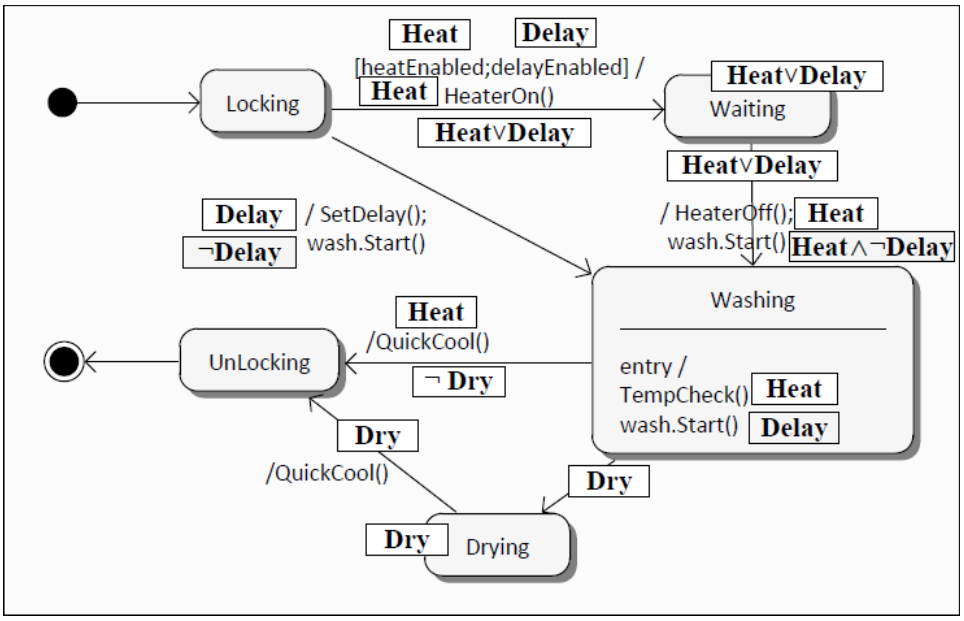


Figure 2.5: Domain model resulting from applying Rule 1 in Figure 2.2 to the SPL in Figure 2.1

In the paper that describes the Lifting algorithm [11], the correctness criteria of the Lifting algorithm are discussed, formalized and defined as seen in Definition 2.4.1. This specifies that the products derived from a product line transformed with the lifted transformation rule, should be the same as if the transformation rule was applied separately to each product. In this definition, both sets of products should be equivalent rather than equal. The paper also shows that the Lifting algorithm satisfies the correctness criteria defined in Definition 2.4.1.

Definition 2.4.1 (Lifting correctness criteria)**.** *Let a rule R and a product line P be given. $R^{\uparrow}$ is a* correct lifting *of R iff (1) for all* rule applications $P \overset{R^{\uparrow}}{\Longrightarrow} P'$, $\text{Conf}(P') = \text{Conf}(P)$*, and (2) for all configurations ρ in* $\text{Conf}(P)$*, $M \overset{R}{\Longrightarrow} M'$, where M is derived from P, and M' is derived from P' under ρ.*

2.5 Coupled changes of feature models and domain models

To develop a software product line, both the feature and domain models must be updated. This is necessary to turn any product line into any other product line. Product lines can be extended by adding new features or removing deprecated features. These changes lead to products being created or removed. To allow for these changes, coupled changes of the feature and domain model are required.

An earlier work [14] explored the possibility to express and execute coupled changes of the domain model and feature model by means of graph transformations. This work proposed a formal construction of product line pushouts, which can be used for supporting these changes. The formal construction is designed to have lifting as a special case, in which the change of the feature model is empty. Yet, this earlier work does not yet provide an efficient algorithm for performing the changes enabled by the formal construction. The main performance bottleneck is a certain check that considers all products of the input product line individually, which, for the reasons discussed earlier, is prohibitively expensive for large product lines.

2.6 Knowledge gap

None of the works considered so far allows someone to efficiently transform whole software product lines with all involved artifacts: The Lifting algorithm introduced in Sect. 2.4 only supports changes of the domain model. The Lifting algorithm assumes that the feature model of the SPL is constant. As the algorithm does not manage changes in the feature model, a feature in an SPL cannot be removed using the Lifting algorithm nor can a feature be added to an SPL. A new feature that adds a new state would need the state to have the presence condition of the new feature. The Lifting algorithm only changes the products for which existing features match and it cannot create new products.

The construction of product line pushouts from Sect. 2.5 relies on checking certain conditions for all products of the product line, and thus, requires an explicit enumeration of products. To use software product lines in a real setting, we need an algorithm to apply graph transformation rules to software product lines without enumerating all products.

A new algorithm must be able to apply transformation rules that affects the domain model as well as the feature model, changing both in a consistent way.

Chapter 3

Research

In this thesis, we develop an algorithm that generalizes and improves upon the Lifting algorithm [11] by supporting coupled changes of the feature model, the domain model, and the presence conditions.

To support these changes, we need a suitable representation of rules. We will define a notion of an augmented transformation rule and show examples of transformation rules that make use of the support for the coupled changes in the new algorithm. These transformation rules have a new well-formedness condition and a few assumptions made by the algorithm.

Next, we will explain how the algorithm manages to support the application of these transformation rules, and we define the algorithm itself. We will discuss its correctness and complexity.

Finally, we present a Python implementation of the new algorithm in an isolated environment.

3.1 Rule representation

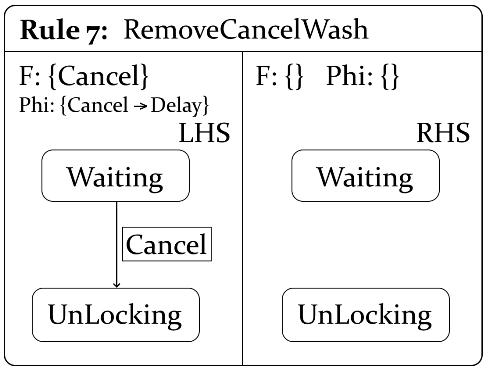


Figure 3.1: Transformation rule that removes a Cancel feature from the domain model and the feature model of a software product line

We need a rule representation that allows the user to specify coupled changes of the feature and the domain model. Conventional rules (Def. 2.2.1) are not suitable for this purpose, since they only address changes of the domain model. To also support changes of the feature model and its mapping to the domain model, we introduce augmented rules. For example, the rule 7 in Figure 3.1 is an augmented rule that defines how a feature *Cancel* is removed where both the domain model and the feature model are changed.

The following definition supports the specification of augmented rules by defining the LHS and the RHS as software product lines. As a product line, the LHS and the RHS can include a feature model with features F and an extra set of constraints Φ. With these, a transformation rule can specify changes made to the feature model.

Definition 3.1.1 (Augmented transformation rule)**.** *An* augmented transformation rule *R is a tuple $R = \langle \{NAC\}, LHS, RHS \rangle$, where LHS and RHS are the software product lines called the* left-hand *and the* right-hand *sides of the rule, respectively, and $\{NAC\}$ represents a (potentially empty) set of software product lines called the* negative application conditions.

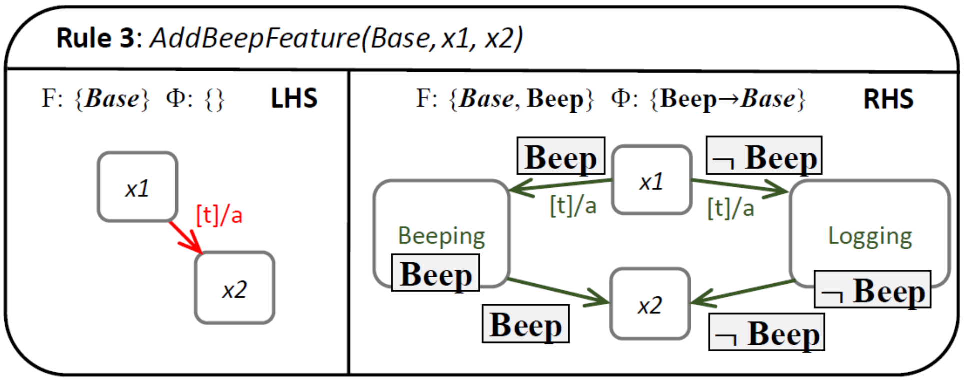


Figure 3.2: Transformation rule for adding the feature *Beep* to a software product line

Feature Model

F: { **Wash**, **Heat**, **Delay**, **Dry**, **Beep** }

Φ : {**Wash**, **Delay**→¬**Heat**, **Beep**→ **Wash** }

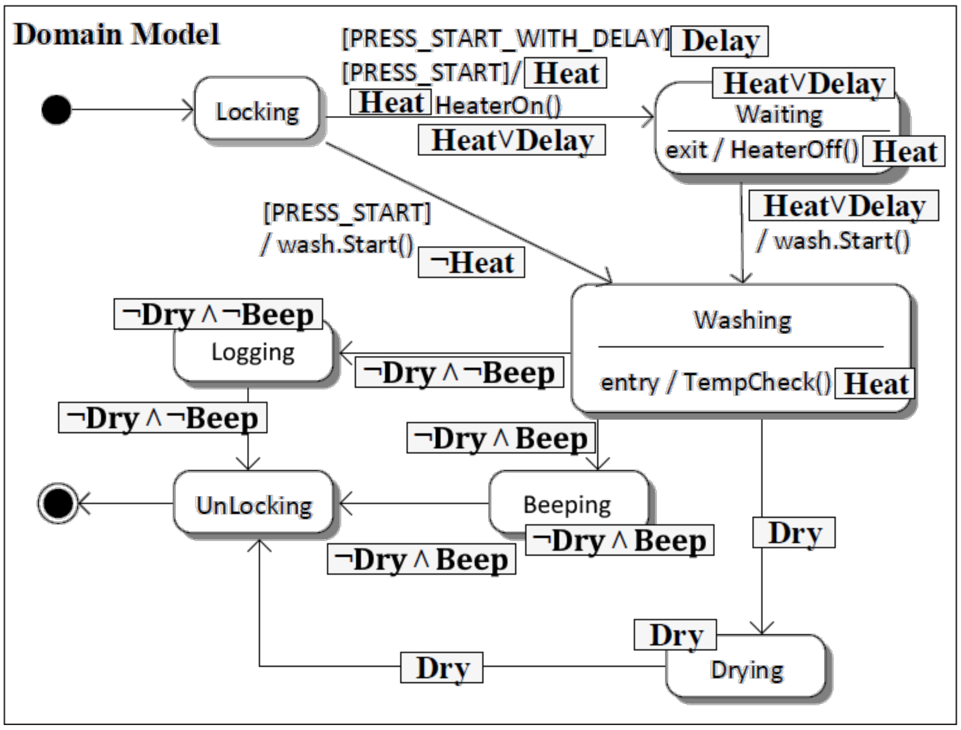


Figure 3.3: Washing Machine Controller Product Line after application of Rule 3 in Figure 3.2

Rule 3 in Figure 3.2 shows the proposed rule for adding a new feature *Beep* to a software product line [14]. Features can be added together with new nodes and edges for the domain model. In the rule, these nodes and edges can have a presence condition that differentiates the set of products with the new feature from the set of products without the new feature. Here, the unrounded boxes, show that the Beeping node, along with its connecting edges, should only be present in the products with the Beep feature, whereas the Logging node along with its connecting edges, should only be present in products without the Beep feature. Note that the feature model changes in the same transformation. The feature *Beep* is added to the feature model, and a condition in Φ is added that represents the dependency of the new feature, on a base feature from the input product line. After application of this rule to the SPL in Figure 2.1, the resulting SPL should be equivalent to the SPL in Figure 3.3.

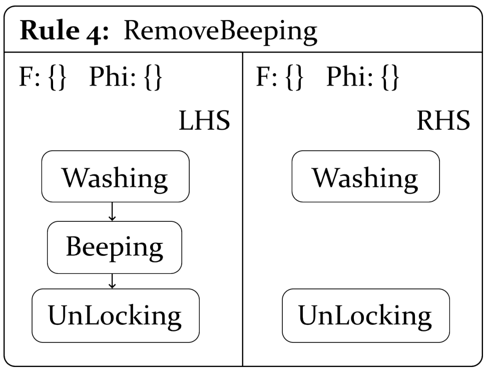


Figure 3.4: Transformation rule for removing the Beeping node and edges

Rule 4 in Figure 3.4 is the start for the transformation rules needed to reverse adding the *Beep* feature after application of rule 3 in Figure 3.2. It removes the Beeping node and the edges that connect to it from the software product line, by removing them from the products that have them.

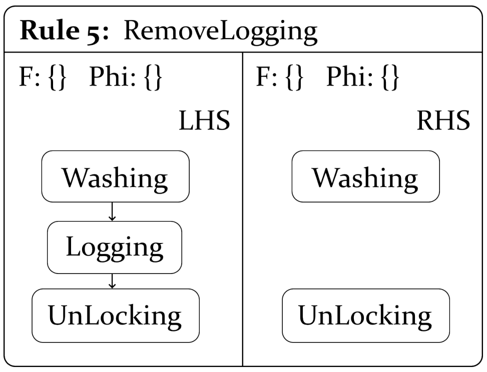


Figure 3.5: Transformation rule for removing the Logging node and edges

Rule 5 in Figure 3.5 is the second step needed to reverse adding the *Beep* feature after application of rule 3 in Figure 3.2. It is very similar to rule 4 in Figure 3.4. However, it removes the Logging node and the edges that connect to it, rather than the Beeping node and its edges.

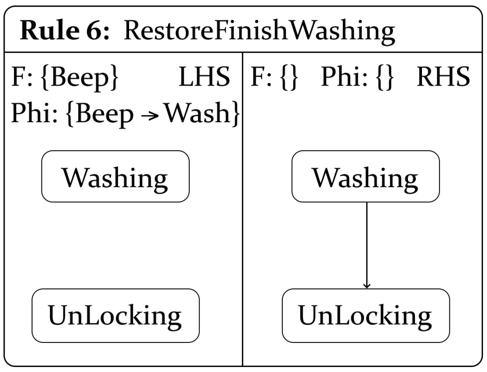


Figure 3.6: Transformation rule to restore the edge from Washing to UnLocking and finally remove the Beep feature

After application of rules 4 and 5, the feature *Beep* can be removed, and the edge from Washing to UnLocking can be restored as defined by rule 6 in Figure 3.6.

3.1.1 Well-formedness condition

The augmented transformation rule has an important condition that needs to be satisfied for the transformation rule to be well-formed:

A transformation rule cannot have any features in the PCs on the RHS that are removed by the rule, or not present in the software product line.

Without this rule, elements in the domain model might only be present in products with a feature that is not in the SPL's feature model. This would result in the output of an invalid SPL.

3.1.2 Assumption

Some transformation rules might be considered well-formed, but are either not supported by our algorithm or should not be applied to some product lines. There is one important assumption made of a rule:

The union of all presence conditions on the LHS must be satisfiable. The new algorithm is based on the Lifting algorithm, and cannot modify the SPL if this does not hold, which is not the desired effect. An example of a rule not fulfilling this condition is Rule 2 in Figure 3.7. This is because no products exist such that R and $\neg R$ holds for their features. Allowing changes to the feature and domain model such as those specified by Rule 2 are desirable, but out of scope for this thesis.

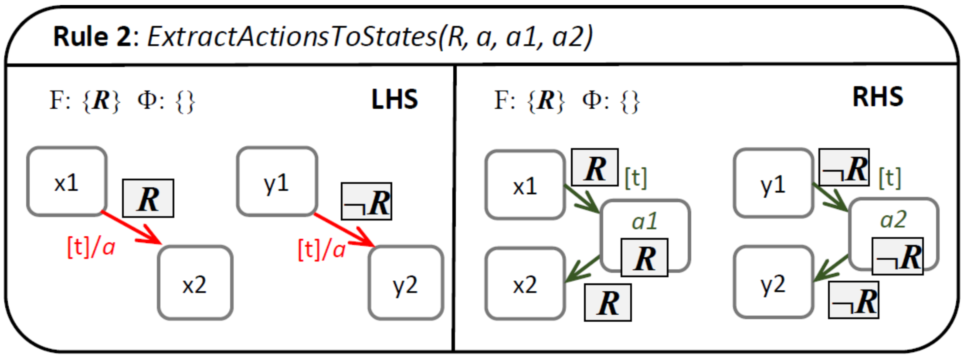


Figure 3.7: A transformation rule to extract actions to states

3.1.3 Validity of matches

A transformation rule can only remove a feature from the feature model, if and only if the following conditions hold:

1. All uses of the feature in a constraint within the feature model are removed by the rule application as well.
2. All uses of the feature in a presence condition within the domain model are removed by the rule application as well.

Each feature, except for the root feature has a dependency on another feature. Because this is encoded as a constraint in Φ, always at least one constraint has to be removed when a feature is removed. These two constraints form a "removal-condition". Without this condition, a transformation rule can remove a feature f_{rem} from the feature model, while an element in the domain model still has f_{rem} as the presence condition, which is now nonexistent. Because of this, sometimes multiple transformation rules are required to remove features from the feature model.

3.2 Algorithm

Our algorithm for coupled changes extends the Lifting algorithm introduced in Section 2.4. Lifting is an algorithm for applying transformation rules to software product lines, focusing on changes of the domain model. As the Lifting algorithm is not geared to support changes of the feature model, changes have to be made to the algorithm to be able to add and remove features, and to deal with changes of elements' presence conditions. This section discusses the supported types of changes and defines the final algorithm. Since the new algorithm is an extension of the Lifting algorithm, from now on, we will call it the Extended Lifting algorithm.

The Extended Lifting algorithm assumes an augmented transformation rule as input as defined in 3.1.1. To allow a non-augmented rule to be applied, the LHS and the RHS of the rule must be turned into product lines, by adding empty feature models to both.

3.2.1 New algorithm

Figure 3.8 shows the pseudo code of the new algorithm, that is used to apply the augmented transformation rules to product lines.

```
Data: Product line $P$ with constraint $\Phi_P$, a valid augmented rule
      $R$, matching site $K = \langle \bar{N}, C, D \rangle$ in the domain model of P
Result: Transformed product line $P'$
1  $P' = P$
2  $\Phi_{apply} = \neg \bigvee \{\phi_N^{and} | N \in \bar{N}\} \wedge \phi_C^{and} \wedge \phi_D^{and}$
3  if $\Phi_P \wedge \Phi_{apply}$ is SAT then
4      add features and constraints from the feature model on the RHS
         that are absent in the feature model on the LHS to the feature
         model of the $P'$
5      remove features and constraints from the feature model of $P'$
         which are present in the feature model on the LHS, but absent
         in the feature model on the RHS.
6      for $a \in A^r$ do
7          add $a$ to domain model of $P'$
8          if $PC_a^R$ is not empty then
9              $\phi'_a = PC_a^R \wedge \Phi_{apply}$
10         else
11             $\phi'_a = \Phi_{apply}$
12         end
13     end
14     for $c \in C$ do
15         if $PC_c^R$ is not empty then
16             $\phi'_c = \phi'_c \wedge PC_c^R$
17         end
18     end
19     for $d \in D$ do
20         $\phi'_d = \phi_d \wedge \neg\Phi_{apply}$
21         if $\Phi_P \wedge \phi'_d$ is not SAT then
22             remove $d$ from domain model of $P'$
23         end
24     end
25 end
26 return $P'$
```

Figure 3.8: The new Extended Lifting algorithm

3.2.2 Lifting comparison

The Extended Lifting algorithm is very similar to the Lifting algorithm, but there are a few key changes to allow for coupled transformation of domain and feature models. Φ_{apply} is calculated in line 2 and the conjunction $\Phi_P \wedge \Phi_{apply}$ is checked for satisfiability in line 3, the same as with the Lifting algorithm.

Line 4 and 5 are new, and the only lines that can change the feature model of the product line. Line 4 handles the creation of new features and constraints by comparing the feature models of LHS and the RHS in the rule. In the augmented transformation rule, the LHS and the RHS are defined as product lines, which is necessary to define the feature models. When a new feature is added, a constraint is added to show the dependency of the new feature on a base feature. This base feature is mapped to a feature in the feature model and this mapping is part of the matching site.

The for-loop in line 6 - 13 manages added elements to the domain model. The Lifting algorithm loops through the elements to be added, and adds them to the domain model on the next line, and sets the presence condition ϕ'_a to Φ_{apply}. The Extended Lifting algorithm does the same in lines 7 and 11. However, the rule can specify a presence condition for the new element to constrain it to products with specific features. On line 8 it checks if any PC is specified in the rule by checking if it is not empty. In the case it is not, the new PC will be set to the conjunction of the PC specified by the rule, and ϕ_{apply}.

Deletion of elements is handled in the for-loop on lines 19 - 24 and is identical to the Lifting algorithm.

The Extended Lifting algorithm supports the specification of a presence condition to preserved elements to modify the domain model, similar to the PCs an augmented transformation rule can define for added elements. Lines 14 - 17 handle these changes. Any preserved element c for which a PC_c^R is defined in the rule, the PC is constrained to only the products that satisfy the original PC, as well as PC_c^R.

We will now discuss how the algorithm supports several scenarios of coupled changes: adding features, removing features, and changing presence conditions.

3.2.3 Adding features

When a new feature F_{new} is added, the transformation rule can specify which nodes and edges are present in the set of the products with the F_{new}, and which in the set of products without feature F_{new}. Of course, a new node or edge can be added to both as well. This is defined by the presence condition in the rule. Their PC will become the conjunction of Φ_{apply} and the PC, rather than Φ_{apply}, provided that the PC is not empty. In this case the PC is updated on line 9.

When the PC in the rule is empty for a new node or edge, they are added to all products that match Φ_{apply} in on line 11. The PCs can include any of the features in F on the RHS of the transformation rule. If an PC in the rule includes a feature that is not in F on the RHS, the transformation rule is invalid.

Finally, the feature model is changed according to the transformation rule. Any features in the RHS that are not in the LHS are added to the feature model, and the same is done for the elements in Φ. The *Base* feature as used in Rule 3 in Figure 3.2 is matched to a base feature.

3.2.4 Removing features

A transformation rule can specify that a feature F_{rem} has to be removed by having F_{rem} in F in the domain model on the LHS, but not on the RHS. Any features in the feature model on the LHS that are not in the feature model on the RHS, have to be removed from the feature model of the SPL. For example, rule 7 in Figure 3.1 removes the *Cancel* feature. It deletes the edge from the Waiting state to the UnLocking state and also removes the *Cancel* feature from the feature model as this is present in feature model on the LHS, but absent in the feature model on the RHS.

Preparing the domain model

Due to our assumptions (Section 3.1.2), a given rule might not be applicable to an input product line right away. In such cases, it might still be possible to prepare the input product line to make the rule applicable. Specifically, feature F_{rem} can only be removed if the SPL is "F_{rem} removal ready", which is a condition to be checked for a particular match (comparable to the "dangling condition" during the application of a normal graph transformation rule), also referred to as the "removal-condition". This is the case if none of the nodes and edges have the feature F_{rem} in their PC. In other words, the set of products with the feature is equal to the set of products without the feature. Any transformation rule that specifies that feature F_{rem} has to be removed that is matched against an SPL of which the domain model is not "F_{rem} removal ready", cannot be applied.

A number of transformation rules might be necessary to make sure the domain model is F_{rem} removal ready. Figure 3.3 shows the washing machine controller model resulting from adding the *Beep* feature described in Rule 3 in Figure 3.2. If we want to undo the transformation, we will need to remove the *Beep* feature. The first steps to make the domain model *Beep* removal ready, would be to remove the Beeping and Logging states, along with their edges, since their PCs are *Beep* and $\neg Beep$ respectively. This is done by applying the transformation rules 4 and 5 from Figure 3.4 and Figure 3.5 respectively. Now, the resulting SPL is Beep removal ready.

Remark: It is not possible to remove both the Beeping and Logging states using a single transformation rule, because the LHS cannot possibly have any valid matches, as not a single product could satisfy the PCs *Beep* and $\neg Beep$ at the same time.

Since the SPL is is now Beep removal ready, we need one more transformation rule that adds the edge from the Washing node to the UnLocking node which was removed using Rule 3 in Figure 3.2. This final transformation rule is described in Figure 3.6

3.2.5 Adding and removing constraints

The feature model of a product line contains a set of constraints Φ over the features that a product can have. The LHS and the RHS in an augmented transformation rule can have a feature model, which includes these constraints as well. Deletion and addition of constraints can be specified by having each constraint exclusively in the feature model on one side of the rule. However, since these changes don't concern the domain model at all, the concept of "feature removal ready" does not apply.

3.2.6 Changing presence conditions

As explained in Sect. 3.2.2, the domain model of a product line can be modified by specifying a presence condition for preserved elements. As the new PC for such an element is the conjunction of its old PC, and the PC specified by the rule, the products in which the element is present is restricted. In Figure 3.8 on line 15, the algorithm checks if such a PC is specified by the rule. In the case that it is not, the PC of the preserved element is updated on line 16. This can be used to introduce a new feature that adds elements to products that used to be in all products.

3.3 Correctness and complexity

3.3.1 Correctness

With the new transformation rules, the number of products can change, which means that we will need to handle the correctness differently from the Lifting algorithm's correctness criteria in Figure 2.4.1.

The Lifting algorithm can only apply transformation rules without changes to the feature model, and without explicit changes to presence conditions. The Extended Lifting algorithm can apply all of these transformation rules as well, which makes these a special case. With these constraints on the transformation rules, line 4 - 5 in the Extended Lifting algorithm in Figure 3.8, as well as lines 14 - 17 will not make any changes to P', because the feature model is constant, and because PC_c^R is always empty for all elements $c \in C$. Because PC_a^R is always empty as well (no explicit presence conditions are allowed), the if-statement on line 8 will always lead to the second case on line 11. For these cases, the algorithm is clearly the same as the Lifting algorithm in Figure 2.4 and should therefore also be correct.

When a transformation rule modifies the feature model, products can be created and/or removed. The products that are preserved should change as described by the transformation rule. If a product that is preserved satisfies $\Phi_P \wedge \Phi_{apply}$, it should have all the elements added in A^r as executed in lines 6 - 13 in the Extended Lifting algorithm. Any elements $c \in C$ for which PC_c^R is not empty, should be removed, as executed in lines 14 - 18 in the Extended Lifting algorithm, if and only if PC_c^R does not hold according to the configuration of the product. Finally, the product should also not have any of the elements $d \in D$. The Extended Lifting algorithm removes these in lines 19 - 24 just as the Lifting algorithm does.

When products are created, a new feature must be introduced, as it is not possible to create new products without any new features. Any new products are the same as the products without the new feature in the original SPL, with a few changes made to them according to the transformation rule. The rule can specify to add elements only present in the products with the new feature. These elements have to be present in the resulting products with the new feature that satisfy the presence condition as defined in the rule for the element along with Φ_{apply}. These products can still have some elements specifically removed as well, defined by the transformation rule as the elements $d \in D$. Just as how preserved products can have elements c removed specifically by defining PC_c^R on the RHS of the transformation rule, this is also possible for newly created products. Any elements c for which PC_c^R is not empty are removed from the product if the product does not satisfy $\phi'_c \wedge PC_c^R$.

Finally, products can be completely removed, if one of the features that it has is removed from the feature model of the SPL. In the Extended Lifting algorithm, the products with the removed feature and the products without it must be identical before it is removed. This is called "feature removal ready" and thus the domain model of the SPL will not show any changes when a feature is removed. Note that a feature can be removed within the last transformation rule that makes the SPL feature removal ready as no references to the feature can be present in presence conditions in the domain model after application of the rule.

The products from the rule and the matching site are unified and therefore the presence condition of a preserved element will be the conjunction of both the PC in the matching site, and the rule. This is supported by an existing framework based on category theory [14]. The resulting conjunction $\phi'_c \wedge PC^R_c$ can be seen in line 16 in Figure 3.8.

3.3.2 Complexity

The Extended Lifting algorithm, just like the Lifting algorithm, uses a SAT solver. Solving a SAT problem is NP-complete. The number of variables used for SAT solving depend on how many elements are matched on the LHS of the rule, and the variables used in their presence conditions, as well as the number of elements and variables in the presence conditions of the NACs.

The changes to the feature model are very minimal as in general, only one feature or constraint is add or removed. Line 4 and 5 of the Extended Lifting algorithm in Figure 3.8 are linear in the number of added and removed features and constraints.

Since the if-statements and the updates to presence conditions, or the domain model are constant, the main contributors to the complexity are the for-loops at lines 6, 14 and 19. These are all linear in the number of elements added, preserved, and deleted respectively. The algorithm does not have any nested loops, and all elements besides the SAT solving are linear in time complexity, which means that the complexity is mainly affected by SAT solving, and is not any higher than that of the Lifting algorithm. While solving a SAT problem is NP-complete, the complexity scales up nicely as state-of-the-art SAT solvers can handle a million of variables and several millions of constraints efficiently [10]. This is more than sufficient for even very large software product lines. In the literature, the largest software product lines reported on so far include 15,000 features [7].

3.4 Implementation

To help show what the algorithm might look like in practice, we created an implementation in Python. In this implementation, the algorithm is used to apply transformation rules in an isolated environment. The code contains three tests, each in a separate file:

1. The *Adding Beep Feature* test, which contains Rule 3 as seen in Figure 3.2.

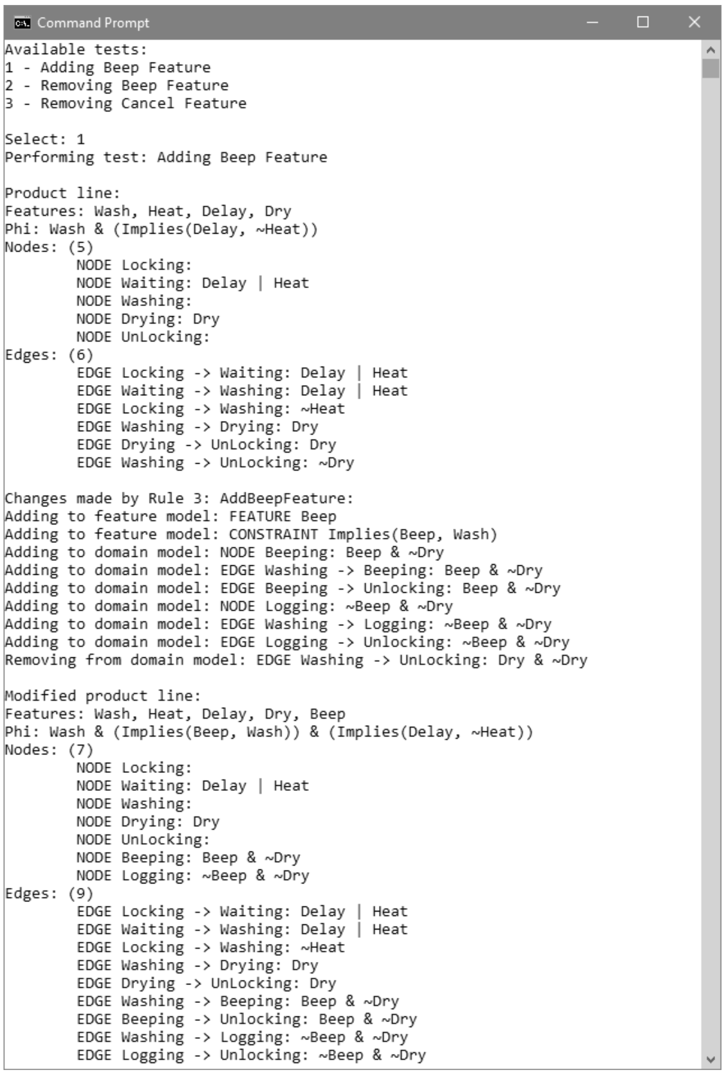
2. The *Removing Beep Feature* test has a collection of rules to remove the Beep feature from the product line in Figure 3.3. These are Rule 4 (Figure 3.4), Rule 5 (Figure 3.5) and Rule 6 (Figure 3.6).

3. The *Removing Cancel Feature* test, which contains Rule 6 from Figure 3.1 that removes the Cancel feature from a product line.

The Python code is split into several files dedicated to different tasks. All files are included in an online appendix [9].

Figure 3.9 shows how the implementation applies rule 3 from Figure 3.2 to a software product line. When performing a test, it shows the initial product line first, and for each rule applied afterwards, it shows the changes made to the product line, and the intermediate/final product line.

The implementation is dependant on the “Sympy” library for performing satisfiability tests, as well as displaying expressions.



Command Prompt

```
Available tests:
1 - Adding Beep Feature
2 - Removing Beep Feature
3 - Removing Cancel Feature

Select: 1
Performing test: Adding Beep Feature

Product line:
Features: Wash, Heat, Delay, Dry
Phi: Wash & (Implies(Delay, ~Heat))
Nodes: (5)
        NODE Locking:
        NODE Waiting: Delay | Heat
        NODE Washing:
        NODE Drying: Dry
        NODE UnLocking:
Edges: (6)
        EDGE Locking -> Waiting: Delay | Heat
        EDGE Waiting -> Washing: Delay | Heat
        EDGE Locking -> Washing: ~Heat
        EDGE Washing -> Drying: Dry
        EDGE Drying -> UnLocking: Dry
        EDGE Washing -> UnLocking: ~Dry

Changes made by Rule 3: AddBeepFeature:
Adding to feature model: FEATURE Beep
Adding to feature model: CONSTRAINT Implies(Beep, Wash)
Adding to domain model: NODE Beeping: Beep & ~Dry
Adding to domain model: EDGE Washing -> Beeping: Beep & ~Dry
Adding to domain model: EDGE Beeping -> Unlocking: Beep & ~Dry
Adding to domain model: NODE Logging: ~Beep & ~Dry
Adding to domain model: EDGE Washing -> Logging: ~Beep & ~Dry
Adding to domain model: EDGE Logging -> Unlocking: ~Beep & ~Dry
Removing from domain model: EDGE Washing -> UnLocking: Dry & ~Dry

Modified product line:
Features: Wash, Heat, Delay, Dry, Beep
Phi: Wash & (Implies(Beep, Wash)) & (Implies(Delay, ~Heat))
Nodes: (7)
        NODE Locking:
        NODE Waiting: Delay | Heat
        NODE Washing:
        NODE Drying: Dry
        NODE UnLocking:
        NODE Beeping: Beep & ~Dry
        NODE Logging: ~Beep & ~Dry
Edges: (9)
        EDGE Locking -> Waiting: Delay | Heat
        EDGE Waiting -> Washing: Delay | Heat
        EDGE Locking -> Washing: ~Heat
        EDGE Washing -> Drying: Dry
        EDGE Drying -> UnLocking: Dry
        EDGE Washing -> Beeping: Beep & ~Dry
        EDGE Beeping -> Unlocking: Beep & ~Dry
        EDGE Washing -> Logging: ~Beep & ~Dry
        EDGE Logging -> Unlocking: ~Beep & ~Dry
```

Figure 3.9: Screenshot of the implementation applying rule 3: AddBeepFeature as defined in Figure 3.2

Chapter 4

Related Work

As discussed in chapter 1 and 2, existing work has researched graphs [2], what they can be used for [8], how they can be transformed, and how this applies to software product lines [14, 4, 1, 3]. The Lifting algorithm [11] can apply transformation rules to software product lines. What is missing, and what this paper focuses on, is the application of transformation rules on software product lines without constant feature models. An earlier work has thoroughly explored this possibility on a theoretical level [14], which is necessary to apply software product lines in a real setting. This work introduces a notion of product-line pushouts and uses it to formally define the semantics of the rule applications that affect both the feature model and the domain model. Product-line pushouts can be seen as a rule application algorithm, but they require a certain check to be performed to all products individually, which is infeasible for large product lines. Since application of rules that apply coupled transformation of feature and domain model, without enumeration of all products could not be done before, there is no algorithm to compare our algorithm against.

Another work extends the Lifting algorithm in a different direction, towards supporting transformation rules with variability in them [12], using a variability representation from another work [13]. However, like the Lifting algorithm itself, this work does not support changes of the product line's feature model.

Chapter 5

Conclusions

The Extended Lifting algorithm as seen in Figure 3.8 for applying transformation rules to software product lines is able to apply all the transformation rules with a constant feature model, just as the Lifting algorithm does. On top of that, it allows for changes in the feature model to allow features to be added to or removed from the feature model.

This algorithm is the first to be able to apply these transformation rules to software product lines to allow software product lines to be changed into any other software product line without requiring an explicit enumeration of all products.

We have reasoned about the correctness of the Extended Lifting algorithm and explained why the new algorithm does not have a larger time complexity than the Lifting algorithm, that it is based on.

Finally, a simple Python implementation demonstrates how the Extended Lifting algorithm can be used in practice.

Bibliography

[1] S Apel, D Batory, C Kästner, and G Saake. Feature-oriented software product lines: Concepts and implementation, 2013, 308 pages. *URL http://www. springer. com/computer/swe/book/978-3-642-37520-0.*

[2] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In Andrea Corradini, Hartmut Ehrig, Hans Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, pages 402–429, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[3] Marsha Chechik, Michalis Famelis, Rick Salay, and Daniel Strüber. Perspectives of model transformation reuse. In *International Conference on Integrated Formal Methods*, pages 28–44. Springer, 2016.

[4] Krzysztof Czarnecki and Michał Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glück and Michael Lowry, editors, *Generative Programming and Component Engineering*, pages 422–437, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[6] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[7] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is there a mismatch between real-world feature models and product-line research? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 291–302, 2017.

[8] Leen Lambers, Daniel Strüber, Gabriele Taentzer, Kristopher Born, and Jevgenij Huebert. Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In *ICSE'18:*

IEEE/ACM International Conference on Software Engineering, pages 716–727. ACM, 2018.

[9] Steven Maarse. Online appendix for this thesis. `https://figshare.com/articles/software/Implementation_code/14318282/1`, Mar 2021.

[10] Sven Peldszus, Daniel Strüber, and Jan Jürjens. Model-based security analysis of feature-oriented software product lines. In *GPCE 2018: International Conference on Generative Programming: Concepts & Experience*, pages 93–106. ACM, 2018.

[11] R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik. Lifting model transformations to product lines. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 117–128, 2014.

[12] Daniel Strüber, Sven Peldszus, and Jan Jürjens. Taming multi-variability of software product line transformations. In *FASE*, pages 337–355, 2018.

[13] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. Variability-based model transformation: formal foundation and application. *FAC'18: Formal Aspects of Computing*, 30(1):133–162, 2018.

[14] G. Taentzer, R. Salay, D. Strüber, and M. Chechik. Transformations of software product lines: A generalizing framework based on category theory. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 101–111, 2017.