Lifting Model Transformations to Product Lines

Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, Marsha Chechik
Department of Computer Science
University of Toronto, Toronto, Canada
{rsalay, famelis, mjulia, adisandro, chechik}@cs.toronto.edu

ABSTRACT

Software product lines and model transformations are two techniques used in industry for managing the development of highly complex software. Product line approaches simplify the handling of software variants while model transformations automate software manipulations such as refactoring, optimization, code generation, etc. While these techniques are well understood independently, combining them to get the benefit of both poses a challenge because most model transformations apply to individual models while model-level product lines represent sets of models. In this paper, we address this challenge by providing an approach for automatically "lifting" model transformations so that they can be applied to product lines. We illustrate our approach using a case study and evaluate it through a set of experiments.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Theory, Management

Keywords

Software Product Lines, Model Driven Engineering, Model Transformations

1. INTRODUCTION

Model Driven Engineering (MDE) and Software Product Line Engineering (SPLE) are powerful techniques used in industry for managing the complexity of large scale software development. MDE helps manage complexity by using models to raise the level of abstraction at which developers create code. In this context, model transformations are the key enabling technology for automating the movement within and between levels of abstraction [40]. SPLE approaches help manage complexity by treating large sets of similar software product variants as a single conceptual unit rather than a set of individual products, explicitly capturing product line commonalities and variabilities [11].

Both MDE and SPLE are used by numerous industrial organizations, and particularly, in the automotive embedded systems domain. However, combining SPLE with model transformations is a challenging task: most existing transformations, such as model refactoring or code generation, are developed for individual product models and do not take SPLE variability constructs into account.

As a consequence, an organization that relies on SPLE to manage its product portfolio cannot reuse existing third party transformations. Instead, it has to apply them to the individual products derived from the product line or redevelop the transformations in order to apply them to the entire product line. The former approach is impractical because the transformations that need to be applied must be tracked along with the product line and kept up-to-date. Moreover, it is often desired to apply the transformation on the level of the entire product line in order to enable analysis, validation and evolution of the resultant product line. Liebig et al. [29] have shown that analysis applied to the product line outperforms approaches that sample individual products. We give three examples of such techniques below.

1. Classen et al. developed a model checking technique for product lines expressed as transition systems [9]. Using this technique for a product line of, say, UML statecharts, is possible only if the product line is first transformed into an equivalent transition system one. While such a transformation already exists for individual statechart models [46], it does not take variability constructs into account and hence cannot be applied to a product line as a whole.

2. Consider a product line that must support multiple *binding times* – stages of the lifecycle in which decisions about variability are made [43]. Supporting both design and run-time binding requires translating a design-time product line of models into a run-time product line of code. Such a translation can be obtained by *lifting* existing code-generating transformations that are typically used in MDE strategies, to apply to the entire product line instead.

3. Consider the case when a set of standard refactorings has to be performed on a model, e.g., if a new architectural policy requires all public methods in a class have accessor methods. In that case, an existing "Encapsulate Variable" refactoring transformation [31] could be applied to all class diagrams in order to accomplish the task. However, if we want to apply such refactorings to a product line of class di-

ICSE'14, May 31 – June 7, 2014, Hyderabad, India

http://dx.doi.org/10.1145/2568225.2568267

agrams, the transformation would not be directly applicable, and would have to be lifted instead.

While lifting a transformation is useful, doing so manually is hard and error prone because the lifted transformation must correctly address SPLE constructs and consider all possible product variants derived from the product line model. In this paper, we propose an approach for lifting the transformations *automatically*, i.e., no manual changes to the transformation are required to enable it to apply to the entire product line.

Specifically, we make the following contributions: (1) We define and prove correctness of a general lifting algorithm for graph rewriting-based model transformations. The algorithm is designed for the annotative product line approach typically used in practice. (2) We provide a prototype implementation of the algorithm integrated into an existing transformation engine [3]. (3) We use this implementation with a benchmark case study for modeling techniques [7]. (4) Finally, we empirically evaluate the scalability of the algorithm and its implementation. The results suggest that the approach has good scaling behaviour. Note that the focus of our approach is not SPLE-specific transformations (e.g., adding a feature or refactoring a feature model), but rather transformations applicable to individual products.

The rest of the paper is structured as follows. In Sec. 2, we motivate the problem and our solution using a simple product line of washing machine controllers. Sec. 3 provides the needed background on product lines and model transformations. Our approach is presented in Sec. 4. In Sec. 5, we describe the implementation of the lifting approach and present its application to the benchmark case study. Sec. 6 describes a set of experiments aimed to study the scalability of our approach. We compare our approach with related work in Sec. 7 and conclude the paper in Sec. 8.

2. MOTIVATING EXAMPLE

Fig. 1 shows a simple product line W for washing machine controllers expressed using a UML state machine. The *feature model* (the top part of the figure) allows for three optional features to be added to a basic washing machine: (**Heat**) adds the ability to have hot water washes, (**Dry**) adds an automatic dry following the wash, and (**Delay**) adds the ability to delay the start time of the wash. Note that the heated wash and delayed wash features are mutually exclusive while drying can be added independently. The `Excludes` constraint between **Heat** and **Delay** in the feature model indicates that at most one of these can be selected.

The *domain model* of W (the bottom of Fig. 1) is a state machine which specifies that after initiating and locking the washer, a basic wash begins or a waiting period is initiated, either for heating the water or for a delayed wash. Then the washing takes place, followed, optionally, by drying. Finally, if drying or heating was used, the clothes are cooled and the washer is unlocked, terminating the process.

Depending on which of the features have been selected, only some parts of this process may be available. The propositional formulas in boxes throughout the controller indicate the *presence conditions* [12] for different model elements, i.e., the configurations of features under which the element is present in a product. For example, the transition from state `Locking` to state `Waiting` is only present if either feature **Heat** or feature **Delay** is selected; it is guarded by `heatingEnabled` and has action `HeaterOn`() only when feature **Heat** is selected, while it is guarded by `delayEnabled` only if feature **Delay** is selected.

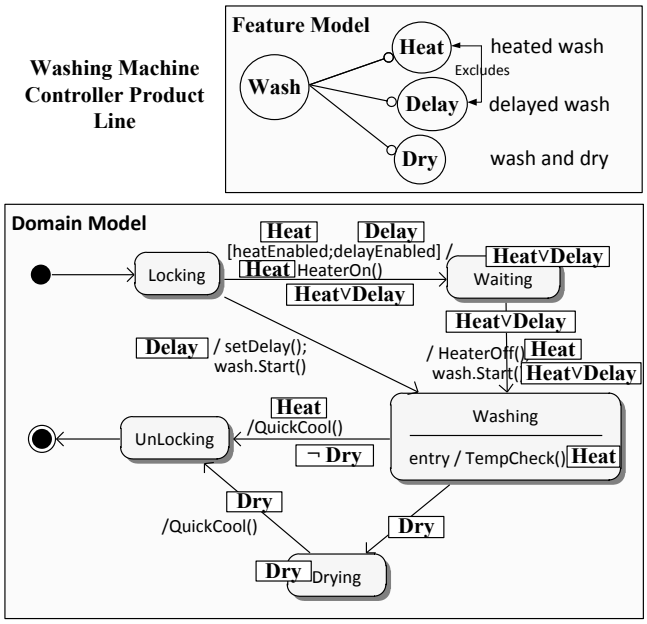

Figure 1: Example washing machine controller product line W.

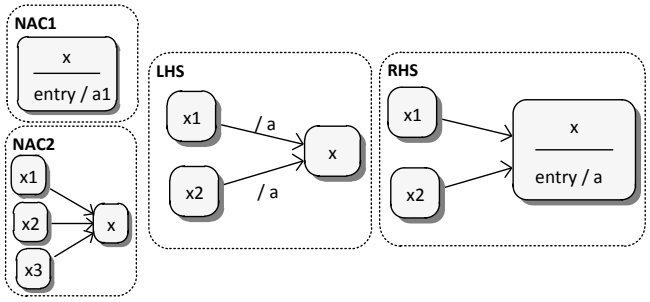

Figure 2: The "fold incoming actions" rule R_F for refactoring a state machine.

Consider a simple state machine transformation. Fig. 2 shows a transformation rule R_F that implements the "fold incoming actions"[1] refactoring transformation that moves common actions on incoming transitions to a state into the entry action for the state. Specifically, the rule is applied to a state machine by attempting to match it to the location where some state, x, has two incoming transitions with a common action, a, as depicted in the LHS of the rule in the middle of Fig. 2. Then the matched portion is replaced with the RHS of the rule (on the right of the figure) which deletes action a from the transitions and makes it the entry action of state x. The *negative application conditions* (NACs, on the left of Fig. 2) prevent the rule from being applied when state x already has an entry action (NAC1) or when there are more than two incoming transitions to it (NAC2)[2]. The transformation is executed by applying the rule R_F to the state machine until it can no longer be applied.

[1]Based on a refactoring by the same name presented in [42].
[2]The general case allows moving the action if it is present in all incoming transitions but we limit it to two transitions for simplicity.

While the above description makes it clear how to apply this transformation to an arbitrary state machine, the goal of this paper is to apply such a transformation to an *entire product line* of models rather than a single product. Thus, we aim to apply the transformation rule R_F to the product line W. Since W represents a set of possible state machine variants, we would expect that the application of R_F to W should act *as if* R_F were applied to each variant separately. However, applying R_F directly to the domain model of W does not achieve the desired result for several reasons:

1. It may miss valid applications of the rule. For example, the two incoming transitions to state `Washing` have the same action `wash.Start()` which would match the LHS of the rule but since state `Washing` already has an entry action `TempCheck()`, the rule does not seem to be applicable because it matches the negative application condition NAC1. However, if we consider the presence conditions of the features, we see that the entry action `TempCheck()` only exists for those products in which feature **Heat** is selected. Thus, the rule *is* applicable for some products, specifically, those in which the feature **Delay** is selected.

2. It may cause inappropriate applications of the rule. For example, R_F seems to be applicable to the two occurrences of action `QuickCool()` on the two incoming transitions to state `UnLocking`. However, if `QuickCool()` were folded into `Unlocking`, it would be present in all products, even those for which neither **Heat** nor **Dry** are selected.

3. The presence conditions may be affected by rule applications. For example, R_F is applicable to the incoming transitions of state `Washing` with action `wash.Start()` when the feature **Delay** is selected. As a result, the action should be deleted from these transitions and added as an entry action to state `Washing` only when **Delay** is selected. This can only be accomplished by setting the presence conditions for these elements appropriately. Yet conventional transformation rules such as R_F do not manipulate presence conditions!

In the rest of this paper, we describe an approach to address these complexities in a generic way in order to adapt rules such as R_F to be correctly applicable to product lines.

3. BACKGROUND

In this section, we fix the notation and provide the necessary background on product lines and model transformations.

3.1 Product Lines

We follow the *annotative* product line approach [12, 25, 35], formally defined below.

Definition 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 $\langle \mathtt{E}, \phi_\mathtt{E} \rangle$ mapping a domain model element* `E` *to a propositional formula $\phi_\mathtt{E}$ over features. The formula $\phi_\mathtt{E}$ is referred to as the* presence condition *of the element* `E`.

For the example in Fig. 1, the feature model of the product line W contains four features: **Wash**, **Heat**, **Delay** and **Dry**. Relationships between these features are defined by the propositional formula $\Phi_W = \mathbf{Wash} \wedge \neg(\mathbf{Heat} \wedge \mathbf{Delay})$.

In this example, domain model elements are state machine constructs such as states, transitions, state entry and exit activities, and transition actions. The presence conditions are given in boxes next to the corresponding domain model elements, e.g., the state `Waiting` in Fig. 1 is annotated by the presence condition **Heat**$\vee$**Delay**. Feature **Wash** is mandatory and thus always occurs. For simplicity of presentation, we omit **Wash** from the presence conditions. We also do not annotate elements whose presence conditions are *true*, e.g., the state `Locking`.

Definition 2 (Feature Configuration). *A* valid feature configuration *ρ of a product line P is a subset of its features that satisfies Φ_P, i.e., Φ_P evaluates to* true *when each variable f of Φ_P is substituted by* true *when $f \in \rho$ and by* false *otherwise. The set of all valid configurations in P is denoted by* $\mathsf{Conf}(P)$.

Definition 3 (Product Derivation). *A product M is* derived from *the product line P under the feature configuration ρ if M contains those and only those elements from the domain model whose presence conditions are satisfied for the features in ρ.*

For the example in Fig. 1, sets {**Wash**, **Heat**, **Dry**}, {**Wash**, **Dry**} and {**Wash**} are some of the valid configurations of the product line W. Any set not containing the feature **Wash** or containing both **Heat** and **Delay** does not correspond to a valid configuration as it violates the formula Φ_W given above. The product derived using only the feature **Wash** will go through the states `Locking`, `Washing` and `Unlocking`, while the product derived using the features **Wash** and **Dry** will go through the states `Locking`, `Washing`, `Drying` and `Unlocking`.

Note that while our work is based on the above definition of annotative product lines it can readily be adapted to other annotative approaches, e.g., CVL [23].

3.2 Model Transformations

In this paper, we focus on model transformations done via *graph transformations* [17]. A graph transformation consists of executing a set of graphical rules defined as follows:

Definition 4 (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.

Fig. 3 depicts the NACs, LHS and RHS of the rule R_F from Fig. 2 as typed graphs using types from the UML metamodel [33]. For example, NAC1 consists of a state `x` with an entry action `a1` that is a UML `behaviour` (e.g., a class operation).

The NACs, LHS, and RHS of a rule consist of different *parts*, i.e., sets of model elements which do not necessarily form proper graphs. These parts play different roles during the rule application:

$\mathbf{C}^r$: The set of model elements that are present both in the LHS and the RHS, i.e., remain unaffected by the rule.
$\mathbf{D}^r$: The set of elements in the LHS that are absent in the RHS, i.e., deleted by the rule.
$\mathbf{A}^r$: The set of elements present in the RHS but absent in the LHS, i.e., added by the rule.

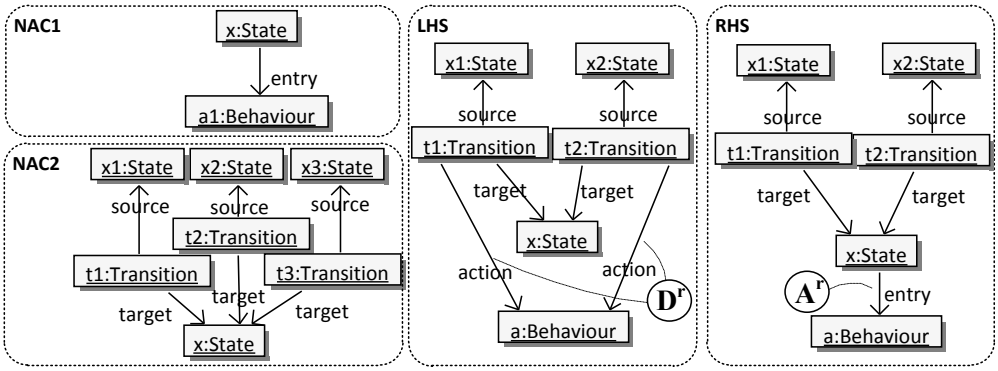

Figure 3: R_F represented as typed graphs.

Table 1: Matching sites of rule R_F in Fig. 3 for the domain model in Fig. 1.

Site	$\overline{\mathbf{N}}$	C	D
K_1	`Washing`, `TempCheck`	`Washing`, `Locking`, `Waiting`, `wash.Start()`, `lw`, `lw_Locking`, `lw_Washing`, `ww`, `ww_Waiting`, `ww_Washing`	`lw_wash.Start()`, `ww_wash.Start()`
K_2		`UnLocking`, `Washing`, `Drying`, `QuickCool()`, `wu`, `wu_Washing`, `wu_UnLocking`, `du`, `du_Drying`, `du_UnLocking`	`wu_QuickCool()`, `du_QuickCool()`

$\mathbf{N}^r$: The set of elements present in any NAC, but not those present in $\mathbf{C}^r$.

For the example rule R_F from Fig. 3, these parts are as follows: $\mathbf{C}^r$ is {`x`,`x1`,`x2`,`a`,`t1`,`t1_x1`,`t1_x`, `t2`,`t2_x2`,`t2_x`}, $\mathbf{D}^r$ is the set {`t1_a`,`t2_a`}, $\mathbf{A}^r$ is the set {`x_a`}, $\mathbf{N}^r$ is set {`a1`,`x_a1`,`x3`,`t3`,`t3_x3`,`t3_x`}. To reduce clutter, only $\mathbf{D}^r$and $\mathbf{A}^r$are explicitly indicated in the figure.

A rule R is *applied* to a model M by finding a *matching site* of its LHS in M:

Definition 5 (Matching site). *A* matching site *of a transformation rule R in a model M is a tuple $K = \langle \overline{\mathbf{N}}, \mathbf{C}, \mathbf{D} \rangle$, where $\mathbf{C}$ and $\mathbf{D}$ are matches of the parts $\mathbf{C}^r$ and $\mathbf{D}^r$ of the LHS of R in M, and $\overline{\mathbf{N}}$ is the set of all matches of NACs in M relative to $\mathbf{C}$ and $\mathbf{D}$.*

Two matching sites for the rule R_F in the washing machine controller in Fig. 1 are shown in Table 1 (two more matches, isomorphic to K_1 and K_2, are not shown for brevity). In this table, `lw` and `ww` are the names of the transitions between states `Locking`/`Waiting` and `Washing`; while `wu` and `du` are the names of the transitions between states `Washing`/ `Drying` and `UnLocking`. The table says, for example, that in part $\mathbf{D}$ of matching site K_1, `t1_a`=`lw_wash.Start()` and `t2_a`= `ww_wash.Start()`.

In the above definition, $\overline{\mathbf{N}}$ denotes the set of all matches within model M of the NACs of R given the match of $\mathbf{C}^r$ and $\mathbf{D}^r$. If the same NAC can match multiple ways, then all of them are included in $\overline{\mathbf{N}}$ as separate matches. For example, if state `Washing` had another input transition, that transition would also appear in $\overline{\mathbf{N}}$ for K_1 since it would match $t3$.

Algorithm: Apply Rule
Input: Rule R, model M, matching site $K = \langle \overline{\mathbf{N}}, \mathbf{C}, \mathbf{D} \rangle$
Output: Transformed model M'
1: $M' = M$
2: **if** $\overline{\mathbf{N}} = \emptyset$ **then**
3: **let A** be a set of fresh elements corresponding to the part $\mathbf{A}^r$ of R
4: add $\mathbf{A}$ to M',
5: remove $\mathbf{D}$ from M'
6: **return** M'

Figure 4: Algorithm for applying a graph transformation rule.

The set of matching sites define those places in the model where the rule can potentially be applied:

Definition 6 (Applicability condition). *Given a transformation rule R, a model M, and a matching site $K = \langle \overline{\mathbf{N}}, \mathbf{C}, \mathbf{D} \rangle$, R is* applicable at K *iff $\overline{\mathbf{N}}$ is empty*[3].

The above definition ensures that the rule can only be applied at a given site if $\mathbf{C}$ and $\mathbf{D}$ are matched and no NAC is matched. For R_F, the matching site K_1 given in Table 1 does not satisfy the applicability condition since $\overline{\mathbf{N}}_1 \neq \emptyset$. On the other hand, no NACs hold in the second matching site, K_2.

The rule application algorithm is given in Fig. 4. The applicability condition is checked in Step 2 and if it satisfied, the rule is applied by adding the elements in $\mathbf{A}$ (Step 4) and deleting the elements in $\mathbf{D}$ (Step 5). For example, applying R_F to K_2 requires the deletion of the action `QuickCool()` from the two transitions because it is contained in $\mathbf{D}$, and the addition of `QuickCool()` as an entry action for state `UnLocking` according to $\mathbf{A}^r$.

We refer to rules such as the ones described above as *classical*, to differentiate them from their *lifted* counterparts which can be applied to product lines.

4. APPROACH

In this section, we describe the process of lifting a transformation rule to apply to product lines. When a classical

[3]The theory of graph transformation requires some additional formal preconditions, most notably, the *gluing condition* [17]. We so not discuss them here for brevity.

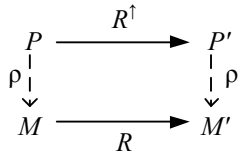

Figure 5: The preservation of configurations to be satisfied by lifting – solid lines denote rule application and dashed lines denote product derivation.

rule R is adapted for product lines, we say that it is *lifted* and denote it by $R^{\uparrow}$.

4.1 Correctness Criteria

We begin by attempting to define the requirements for $R^{\uparrow}$, i.e., how it should act on a product line so that it preserves the effect intended by R. A natural answer is that after applying $R^{\uparrow}$, the target product line should have the same set of products as it would if R were applied separately to each product in the source product line. Furthermore, we would expect that this would also preserve feature configurations. This is illustrated in Fig. 5 – for each configuration ρ, the result should be the same target product M', regardless of whether $R^{\uparrow}$ is first applied followed by the derivation from P', or if ρ is first used to derive M and then R is applied. We capture these criteria formally:

Definition 7 (Correctness of lifting). *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 $P \overset{R^{\uparrow}}{\Longrightarrow} P'$, $\mathsf{Conf}(P') = \mathsf{Conf}(P)$, *and (2) for all configurations ρ in $\mathsf{Conf}(P)$, $M \overset{R}{\Longrightarrow} M'$, where M is derived from P, and M' is derived from P' under ρ.*

Note that this definition is silent on two points. First, it does not require that the target feature model be identical to the source feature model; they just need to be *equivalent*, i.e., have the same set of valid configurations. However, since R is only defined for the domain model, i.e., it does not manipulate features, a reasonable expectation is that it should leave the feature model unchanged. Second, the above definition does not specify exactly how the domain model should change, as long as the set of products is as required. The same set of products can be represented by different domain models and presence condition [36]. A reasonable expectation here is that the domain model should change as little as possible. These "expectations" are not part of the correctness condition since they are not required to preserve the semantics of R; yet, they are "nice to have" properties for an implementation of lifting – see Sec. 4.5.

When applying a graph transformation rule to a model, it is sufficient to find a graph match of the LHS of the rule and then check whether the NACs are applicable. However, our motivating example in Sec. 2 illustrated that applying a rule to a product line is more complicated because not all domain model elements may appear in a given product, and so the rule may apply to some products and not to others. Thus to satisfy the correctness criterion, we must affect only those products in which the LHS is present and the NACs are absent.

4.2 Lifting Algorithm

We now define what it means to apply a lifted rule $R^{\uparrow}$ to a product line in an analogous way to the definition in Sec. 3.2

Algorithm: Apply Lifted Rule
Input: Product line P with constraint Φ_P, rule R, matching site $K = \langle \overline{\mathbf{N}}, \mathbf{C}, \mathbf{D} \rangle$ in the domain model of P
Output: Transformed product line P'

1: $P' = P$
2: $\Phi_{apply} := \neg \bigvee \{\phi^{and}_{\mathbf{N}} | \mathbf{N} \in \overline{\mathbf{N}}\} \wedge \phi^{and}_{\mathbf{C}} \wedge \phi^{and}_{\mathbf{D}}$
3: **if** $\Phi_P \wedge \Phi_{apply}$ is SAT **then**
4: **for** $a \in \mathbf{A}^{r}$ **do**
5: add a to domain model of P'
6: $\phi'_a := \Phi_{apply}$
7: **endfor**
8: **for** $d \in \mathbf{D}$ **do**
9: $\phi'_d := \phi_d \wedge \neg\Phi_{apply}$
10: **if** $\Phi_P \wedge \phi'_d$ is not SAT **then**
11: remove d from domain model of P'
12: **endfor**
13: **return** P'

Figure 6: Algorithm to apply a lifted graph transformation rule.

of the application of rule R to a model. Lifting a rule is not accomplished via higher-order transformation; instead, we change the execution semantics of rule application.

A matching site for $R^{\uparrow}$ is a matching site for R (see Def. 5) in the domain model of P. Applicability is defined as follows.

Definition 8 (Lifted rule applicability condition). *Given a product line P with constraint Φ_P, a transformation rule $R = \langle \{NAC\}, LHS, RHS \rangle$, and a matching site $K = \langle \overline{\mathbf{N}}, \mathbf{C}, \mathbf{D} \rangle$ in the domain model of P, the lifted rule $R^{\uparrow}$ is* applicable at K *iff $\Phi_P \wedge \Phi_{apply}$ is satisfiable, where $\Phi_{apply} = (\neg \bigvee \{\phi^{and}_{\mathbf{N}} | \mathbf{N} \in \overline{\mathbf{N}}\}) \wedge \phi^{\mathbf{and}}_{\mathbf{C}} \wedge \phi^{\mathbf{and}}_{\mathbf{D}}$.*

Here, $\phi^{and}_{\mathbf{C}}$ is the conjunction of the presence conditions for elements in $\mathbf{C}$; similarly for $\phi^{and}_{\mathbf{N}}$ and $\phi^{and}_{\mathbf{D}}$. This definition says that $R^{\uparrow}$ is applicable iff the presence conditions guarantee that at this matching site, the rule R matches in at least one product of P. Specifically, it checks that there exists a product such that all elements of $\mathbf{C}$ and $\mathbf{D}$ are present and not all elements in any NAC match are present.

The general algorithm of a rule application for a lifted rule is given in Fig. 6. In this algorithm, Steps 2-3 check the applicability condition, Steps 4-7 handle elements added by $R^{\uparrow}$, while Steps 8-12 handle elements deleted by $R^{\uparrow}$. Specifically, Step 5 adds each new element to the domain model of P', and Step 6 sets its presence condition to Φ_{apply} since such elements are added only to those products where R was applicable. For deletion, Step 9 sets the presence condition as $\phi_d \wedge \neg\Phi_{apply}$ to guarantee that the element will be absent, i.e., it is deleted "virtually" in the products where R was applicable, and that it remains intact in all other products where it occurred previously. Step 10 checks whether the element is now present in *any* product and if not, it deletes it "actually" by removing it from the domain model.

As with a classical rule system, lifted rules continue to be applied until no rule is applicable.

4.3 Illustration

We illustrate the lifting algorithm by applying the lift $R^{\uparrow}_{F}$ of the rule in Fig. 3 to the example product line in Fig. 1. The result is shown in Fig. 7, with shading indicating changed presence conditions. Recall that $\Phi_W = \mathbf{Wash} \wedge$

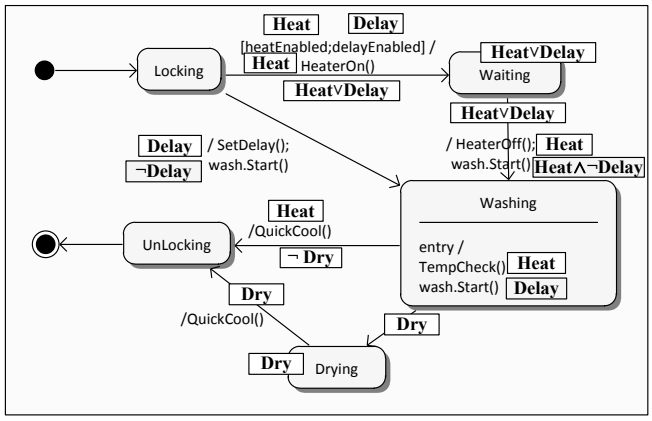

Figure 7: The result of applying the lifted rule $R_F^{\uparrow}$ from Fig. 3 to the product line W in Fig. 1.

¬(**Heat**∧**Delay**) (from Sec. 3.1). The two matching sites for the rule are shown in Table 1. For K_1, Φ_{apply} =**Delay** after substituting the presence conditions from Fig. 1 and simplifying. Thus, the applicability condition $\Phi_W \wedge \Phi_{apply}$ =**Wash**∧ ¬**Heat**∧**Delay**, is satisfied only in those products that have **Wash**, **Delay** and *not* **Heat**. This is because when **Heat** is selected, the entry action `TempCheck`() occurs, and this triggers NAC1, so the rule is not applicable. The only element added in Steps 4-7 is the new entry action `wash.Start`() for state `Washing`, with the presence condition **Delay**. Step 9 virtually deletes the action on the transition out of `Locking` when **Delay** is selected by changing its presence condition to ¬**Delay** while the one out of `Waiting` becomes **Heat**∧¬**Delay**. Step 11, which would really delete these actions, is not triggered for either transition since there are still products that require the actions on these transitions (i.e., Step 10 yields SAT for both). For the matching site K_2, Φ_{apply} =**Dry**∧ ¬**Dry**∧**Heat** which is inconsistent and hence unsatisfiable. Thus, the algorithm does not proceed beyond Step 3 and the rule is not applied.

4.4 Analysis

In this section, we discuss the correctness of the lifting algorithm in Fig. 6 as well as the properties of termination and confluence for lifted rules. These results apply to *arbitrary* graph transformations being lifted and are not dependent on other properties of the transformations such as their being injective, endogenous, exogenous, and so on.

Correctness. Here we show that a transformation lifted according to the algorithm in Fig. 6 applied to product line P to produce P' satisfies the correctness condition in Def. 7.

First note that the lifting algorithm does not manipulate the feature model and so the feature model of P' is identical to the one of P. Thus, $\mathsf{Conf}(P') = \mathsf{Conf}(P)$, and condition (1) is satisfied.

We now show that condition (2) is also satisfied, i.e., for any configuration ρ in P that derives a product M, if R is applied to M to produce M' then M' is the product derived from P' under ρ.

We focus our argument on a specific matching site $K = \langle \overline{\mathbf{N}}, \mathbf{C}, \mathbf{D} \rangle$ since by transitivity, if the rule is correct when applied to each site, then the application to any sequence of sites is also correct. We begin by showing the correctness of the applicability condition (Def. 8) of the lifted rule $R^{\uparrow}$: if R is applicable for product M at site K, then $R^{\uparrow}$ is also applicable, i.e., $R^{\uparrow}$ does not miss any relevant sites.

The condition in Def. 8 says that $\Phi_P \wedge \Phi_{apply}$ must be satisfiable for $R^{\uparrow}$ to apply. But since ρ is a valid configuration, it must satisfy Φ_P by Def. 2 and since, by assumption, R is applicable in M at K, all of the elements in $\mathbf{C}$ and $\mathbf{D}$ are present and no NAC in $\overline{\mathbf{N}}$ is present. Thus, Φ_{apply} holds and $\Phi_P \wedge \Phi_{apply}$ is satisfiable, i.e., the lifted rule applicability condition is correct.

We now argue that the algorithm for lifted rule application, given in Fig. 6, is correct, i.e., applying $R^{\uparrow}$ at site K has the same effect on M as applying R at K in M. We first consider the effect of R on adding elements and then its effect on deleting elements.

Applying R to a product creates new elements according to $\mathbf{A}^r$, and these are the same in every product to which R applies. Thus, Step 5 adds these elements to the domain model. By setting the presence condition for these elements to Φ_{apply} in Step 6 we guarantee that these will be present in all products (including M) where R would have been applicable. Thus, the addition of new elements is correct.

Applying R to a product deletes the elements in $\mathbf{D}$. Step 9 ensures that the presence condition for these elements is further constrained so that they are absent in those products where Φ_{apply} holds. Thus, the deletion of elements is correct. Note that, up to this point, the elements have been only deleted virtually, by limiting their occurrence with presence conditions. Steps 10-11 are an extra "clean-up": these elements are deleted from the domain model if there are no products that contain these elements, given that they now have more constrained presence conditions. Thus, these steps do not affect correctness.

Since both the applicability condition and the effect of rule application (element addition and deletion) are correct, we conclude that $R^{\uparrow}$ is correct w.r.t. Def. 7.

Termination. To prove termination, we show that if an application of a set of classical rules on an input model always terminates, then so does the set of the corresponding lifted rules. Without loss of generality, we restrict ourselves to a rule set containing a single classical rule R which we assume to be terminating. Since $R^{\uparrow}$ is correct according to Def. 7, repeatedly applying it to a product line P has the same effect as repeatedly applying R to each product of P. Since R is terminating, it eventually no longer applies to any product of P. At this point, Φ_{apply} which encodes the classical applicability is *false* and thus $\Phi_P \wedge \Phi_{apply}$ is not satisfiable, and, by Def. 8, $R^{\uparrow}$ does not apply. Thus, when the application of R terminates, the application of $R^{\uparrow}$ terminates as well, i.e., if R is terminating, so is $R^{\uparrow}$.

Confluence. Repeatedly applying lifted rules to a product line P has the same effect as repeatedly applying the corresponding classical rules to each product of P. If the classical rules are confluent and terminating, the process over lifted rules reaches the same final set of products regardless of the order in which rules are applied. Thus, the lifted rule set is confluent "up to equivalence". That is, it always produces product lines with the same set of products.

4.5 Minimality

While they do not affect correctness, issues of minimality of target product line may be relevant to the practical use of the lifting algorithm. We briefly discuss them below.

Domain model minimality. The domain model is not minimal when it contains elements not found in any product. The lifted rule algorithm only affects the presence conditions of added or deleted elements. Steps 10-11 of the algorithm ensure that virtually deleted elements that no longer occur in any product are deleted from the domain model. Added elements have the same presence condition as the rule applicability condition and so these must occur in the products where the rule is applicable. Thus, we conclude that if the domain model was minimal initially then it will remain so after the lifted rule application.

Feature model minimality. A feature is "superfluous" if selecting it does not affect the derived products. A feature model containing superfluous features is not minimal.

Note that the lifted rule application does not affect the feature model part of a product line, and so the source and the target feature models are identical. This is a "nice to have" feature, as discussed following Def. 7. However, the lifted rule application might make some features superfluous. For example, if a transformation was applied to the washing machine controller product line in Fig. 1 that deleted the transition from state `Washing` to `UnLocking` as well as state `Drying` with all the transitions connected to it, then the feature **Dry** would become superfluous.

Minimality of presence conditions. Presence conditions are propositional formulae, and our algorithm does not guarantee that after lifted rule application they will be in a minimal or normal form. While this may affect performance, we do not expect a significant impact on the usability since modelers would rarely look at the presence conditions directly and instead would use tools to manipulate and reason with them.

5. TOOL SUPPORT AND APPLICATION

Tool Support. We implemented the lifting algorithm as an extension to the graph transformation tool Henshin [3], using the Z3 SMT solver [16] to do the SAT checks (algorithm steps 3 and 10). The tool integration platform used was the Model Management Tool Framework (MMTF) [37], an Eclipse-based infrastructure for model management.

The key challenges we faced in the implementation were related to the growth in the number of presence conditions to be tracked at each new step of the lifting algorithm (see Steps 6 and 9). We tackled these using incremental SAT solving techniques.

Application. We used the Car Crash Management software product line case study, referred to as bCMS-SPL [7], as a detailed application scenario for our transformation lifting approach. Our goal was to develop a better understanding of the feasibility of the approach in practical contexts by applying it to a larger, more realistic example. In addition, we compared this example with the results of the scalability study given in Sec. 6.

bCMS-SPL describes a software system for managing the identification, tracking and resolution of car crashes within a community. The system focuses on fire and police as the emergency response providers and facilitates communications between the stakeholders including the victim(s), witnesses, police officers, fire persons, emergency vehicles and coordination personnel. In addition to the basic requirements for such a system, variation points are also specified,

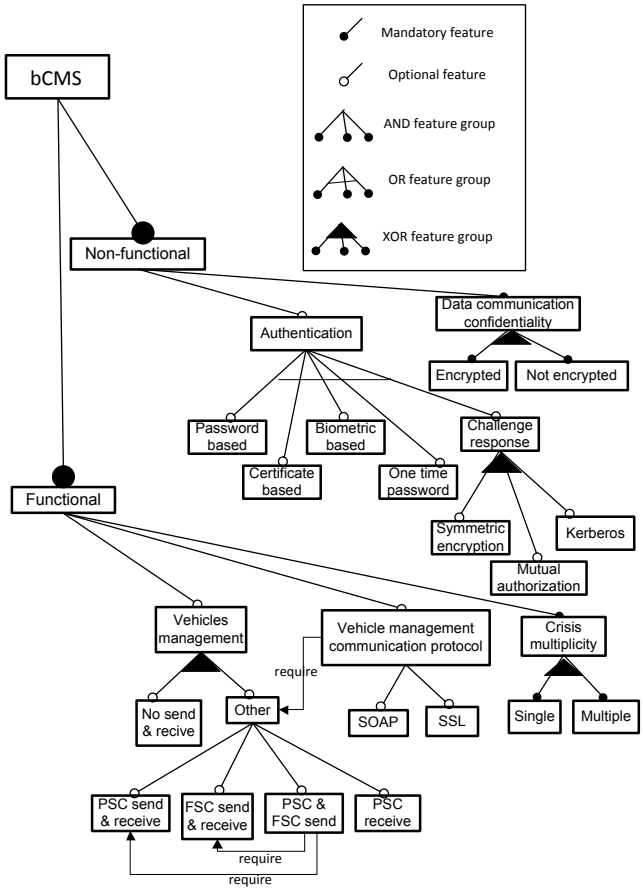

Figure 8: A feature model showing the variation points the bCMS-SPL case study.

shown as a feature model in Fig. 8[4]. These address different ways in which the system could be customized. For example, the feature `CrisisMultiplicity` selects between a system that can only address a single crash at a time and one that can handle multiple crashes in parallel. Overall, the feature model defines 15,360 valid product variants.

We used a documented UML product line developed for the bCMS-SPL requirements [8] and focused on its structural part, i.e., the class diagram. When all of the variants were merged and extended where incomplete, this class diagram had 48 classes, 54 associations, 168 attributes and 174 operations. Of these, 118 had presence conditions (i.e., were not always present).

We applied the lifting to two transformations. The first, a Class Diagram to Relational Database schema translation (C2R), is often used as a benchmark for prototyping and comparing transformation approaches [13]. C2R consists of 9 transformation rules, shown in Fig. 2 of [47], with the LHS parts ranging between 13 and 42 elements. The result of applying the lifted C2R transformation is a product line with the same feature model as in Fig. 8 but applied to an Entity-Relational model. Thus, the C2R transformation illustrates how a lifted transformation can be used to convert a product line for one type of domain model into a product line for another.

[4]Adapted from the feature model in [8] which models five of the seven variation points in bCMS-SPL.

Table 2: Results of experiments with the bCMS-SPL case study.

	Classical		Lifted			
Transformation	Avg Time per App(ms)	Avg # of Apps	Time per App(ms)	Slow down	Max chain	# of Apps
C2R	2.25	357.8	288.60	128.47	3	399
AddGet	0.45	3.6	12.49	27.73	1	5

The second transformation we consider is AddGet – a restriction of the standard class diagram refactoring "Encapsulate Variable" [31] used to add "getter" and "setter" methods to public data members (i.e., attributes) of a class in order to control access to an object's state. For simplicity, AddGet is restricted to just "getter" methods.

Results. We conducted our experiments using our prototype implementation on an Intel Core i7-2600 3.40GHz×4 cores (8 logical) computer with 8GB RAM, running Ubuntu-64 13.04. For each transformation, we ran its lifted and classical versions on five randomly generated products, showing the averaged results in Table 2. The second and third columns show the average running time per rule application, in ms, and the average number of rule applications for the classical rules, respectively; these reflect the fact that C2R is a substantially more complex rule than AddGet.

The sixth column shows the maximum length of rule application "chains" for the lifted rule. A *chain* occurs if the RHS of a rule application is subsequently matched as the LHS or NAC of another rule application. Rule chaining is important since the presence conditions of RHS elements in a rule application are constructed from logical combinations of the presence conditions for LHS and NAC elements (see steps 2, 6 and 9 of the lifting algorithm). Thus rule chains cause presence conditions to grow in size each time a chaining occurs. The maximum length of the chains in both cases is small, and so the growth of presence conditions is not a concern. We use this observation to calibrate the scalability experiments in Sec. 6.

The seventh column shows the number of rule applications for the lifted rule. This number is consistently larger than for the classical rule (column three). We expect the lifted rule to have more applications than the classical rule for two reasons. First, the lifting algorithm can delete an element virtually, via presence conditions (see step 9). Virtual deletion means that the element is still in the domain model and can get matched again in subsequent rule applications. Classical rules delete the element entirely. Second, since the effect of NACs cannot be determined without considering the presence conditions, the transformation engine cannot use the occurrence of NACs in the domain model to eliminate potential rule applications. Thus, this causes more matches than in the classical case. Yet we observed that the impact of these increases is relatively minor.

The fourth column lists the running time for the lifted versions of the rules and the fifth – the slow-down factor. Thus, the lifted version of C2R runs 128 times slower than its classical counterpart whereas AddGet is 28 times slower. Recall that applying the lifted rule is equivalent to applying its classical version to all of the 15,360 products simultaneously! Thus, we conclude that lifting the rules in the bCMS-SPL case study leads to a 2-3 orders of magnitude improvement in performance if compared to the classical application over all products.

6. EVALUATION

In this section, we describe an experimental study aimed to answer the research question *"How does our approach scale with respect to increasing the size of product lines and transformation rules?"* To answer this question, we tried to identify the bottlenecks in our approach and generate randomized inputs to stress test them.[5]

6.1 Methodology

Lifted rule application begins with finding a rule matching site, which is a well-studied subgraph isomorphism problem [18]. Here, our approach performs as well as the traditional graph transformations. Next, the application algorithm in Fig. 6 is applied. Steps 4-9 of the algorithm can be completed in time linear in the size of the matching site K. We thus focus our examination to Steps 3 and 10 since they require solving the satisfiability problem, which is NP-complete. The former entails checking whether $\Phi_P \wedge \Phi_{apply}$ (Def. 8) is SAT and the latter – if $\Phi_P \wedge \Phi_d \wedge \neg\Phi_{apply}$ is UNSAT. In this section, we refer to these formulas as Φ_1 and Φ_2, respectively. Since Φ_1 and Φ_2 are similar in structure, we use the same experimental design whereby we generate random but realistic inputs to a SAT solver to measure the time required to check SAT and UNSAT, respectively. To generate realistic inputs, we simulated the execution of the algorithm in Fig. 6, replacing matching with random element selection. We varied input generation using two experimental variables: (a) the feature model, and (b) the transformation rule. In addition, based on pilot runs and the case study described in Sec. 5, we calibrated random input generation to ensure that the generated formulas correspond to realistic scenarios. We describe the details below.

Varying the Feature Model. Each element in Φ_1 and Φ_2 is represented by its presence conditions which are expressed over the set of features. Moreover, as described in Sec. 3.1, the feature model of a product line P can be encoded in the propositional formula Φ_P expressed over the set of features [15]; Φ_P is a subformula of Φ_1 and Φ_2. Thus, the first experimental variable is the choice of a feature model. To get realistic values of this parameter, we used the collection of real feature models available in the S.P.L.O.T repository [30] . At the time of experimentation, the repository contained 359 real feature models, ranging from 9 to 290 features, with an average of 26 features each.

Varying the Transformation Rule. The subformula Φ_{apply} of Φ_1 and Φ_2 in Def. 8 gets more complex for larger sizes of the rule's LHS and NACs. Thus, our second experimental variable is the choice of the rule. To vary it, we use seven real graph transformation rules chosen from the literature and shown in Table 3. We specifically chose those that represent variety of transformation use cases (translation, refactoring, refinement, etc.) and have LHS, RHS and NACs of different sizes, ranging from 0 to 30.

[5]For more details about the experimental study see `www.cs.toronto.edu/se-research/icse14.htm`

Table 3: Rules used in the experiments. For each rule, the values n, c, d, a are the number of elements in the rule parts $\mathbf{N}^r$, $\mathbf{C}^r$, $\mathbf{D}^r$, $\mathbf{A}^r$, respectively.

#	Rule	Category	n	c	d	a	Source
1	*Relations:StationwMale*	View generation	1	2	1	1	[4], Fig.2
2	UML Activity to Petri Net transition	Refinement	7	2	3	7	[45], Fig.1
3	*while*	Reduction	3	2	8	3	[34], Fig.5
4	*Encapsulate Variable*	Refactoring	30	9	0	24	[31], Fig.20
5	*Fold incoming transitions*	Refactoring	1	12	7	1	[5], Fig.5
6	*attr2fkeyR*	Translation	6	19	0	6	[47], Fig.2
7	*assoc2fkeyR*	Translation	16	21	0	16	[47], Fig.2

Generating inputs. In order to generate realistic inputs, we simulate the rule application algorithm in Fig. 6. At each simulation step r, we produce a formula $\Phi_{apply}(r)$ that approximates the formula Φ_{apply} in Φ_1 and Φ_2. We resorted to simulating the algorithm due to the lack of readily available real examples of product line domain models. We simulated the matching and transformation steps of the algorithm by generating expressions that represent elements with randomly generated presence conditions.

In each simulated rule application we constructed the new presence condition for added or deleted elements from randomly generated presence conditions for the LHS and NACs in the rule. The initial presence conditions were randomly assigned either True or a single feature variable. Then, as new presence conditions were constructed by simulated rule applications, they were put into a pool for possible reuse in subsequent rule applications. This was done to simulate the chaining of the rules (see Sec. 5). In subsequent rule applications, elements were drawn from this pool with a chaining probability and assigned to LHS and NACs of the rule application. The above process was repeated a preset number of times.

Calibrating input generation. The generation process described above requires calibration of a few additional experimental parameters. Rather than considering these as independent variables, we chose to fix their values based on pilot runs and observations of the case study in order to avoid the combinatoric explosion of possible experimental configurations. These parameters are: (1) the size of the original domain model, fixed at 100 elements to simulate models of a reasonable size; (2) the probability that the initial presence condition is a feature rather than True, fixed at 0.6; (3) the maximum number of simulations for each model/transformation rule pair, fixed at 500 rule applications; (4) rule chain lengths, limited to the maximum of 4.

6.2 Results

We implemented the experiment using MMTF [37] as the integration platform and used the hardware setup described in Sec. 5. Each datapoint is obtained by averaging 10 runs.

The results are shown in Fig. 9(a). The horizontal axis uses the logarithmic scale and plots the increase in size of the input feature model, measured by the number of features. The vertical axis plots the time required to check the generated formulas Φ_1 and Φ_2 in seconds. Each rule in Table 3 corresponds to a separate line.

The experiments show that the time required to check the satisfiability of the formulas grows at most linearly for all models, and logarithmically for small to medium sized product lines. Such product lines formed the majority of the samples gathered from S.P.L.O.T. For larger product lines, with more than 100 features, the time increases more linearly.

The determining factor in the observed variation of runtimes is the number of calls to the solver that returned UNSAT (a.k.a. UNSAT calls) – for our examples, these were faster than those returning SAT. Fig. 9(b) shows the number of UNSAT calls against increasing the product line size. The inverse correlation between the solver runtime and the number of UNSAT calls is most dramatic for rule 4 at the 100 feature mark. Similarly, we observe that the "smaller" rules (rules 1-3 in Table 3) have fewer UNSAT calls, take relatively more time to complete and have less variation in processing time. Matching of the "larger" rules (rules 4-7 in Table 3) is more difficult due to their complex patterns and therefore yields more UNSAT calls.

Our preliminary results allow us to conclude that our approach scales well as the size of the product line and the rules increases. The complexity of the problem is likely related to the clause-variable ratio of formulas in the SAT calls and needs further investigation.

Threats to Validity. The first threat to validity is the choice of input models and the random generation of inputs. We attempted to mitigate this by selecting real feature models from S.P.L.O.T and by adapting the random input generation algorithm to closely approximate the real observations from the case study. The second threat is the calibration of the experimental parameters with fixed values, rather than varying them as independent variables. To mitigate this, we calibrated the parameters using values observed in our pilot runs and the case study.

7. RELATED WORK

We have studied the issue of transformation lifting for models with uncertainty [19]. Such models contain annotations to indicate which elements may not exist and use a propositional formula to define sets of elements that can exist at the same time. Although the current paper has been inspired in part by this work, there are substantial differences. Product lines and models with uncertainty both represent sets of models but do so in different ways – the former contain feature models, domain models and presence conditions, while the latter contain only a single model and a single propositional formula, i.e., there is no concept that corresponds to the notion of a feature. As a result of these distinctions, the approach for lifting transformations for product lines does not require expensive propositional formula manipulations needed for models with uncertainty,

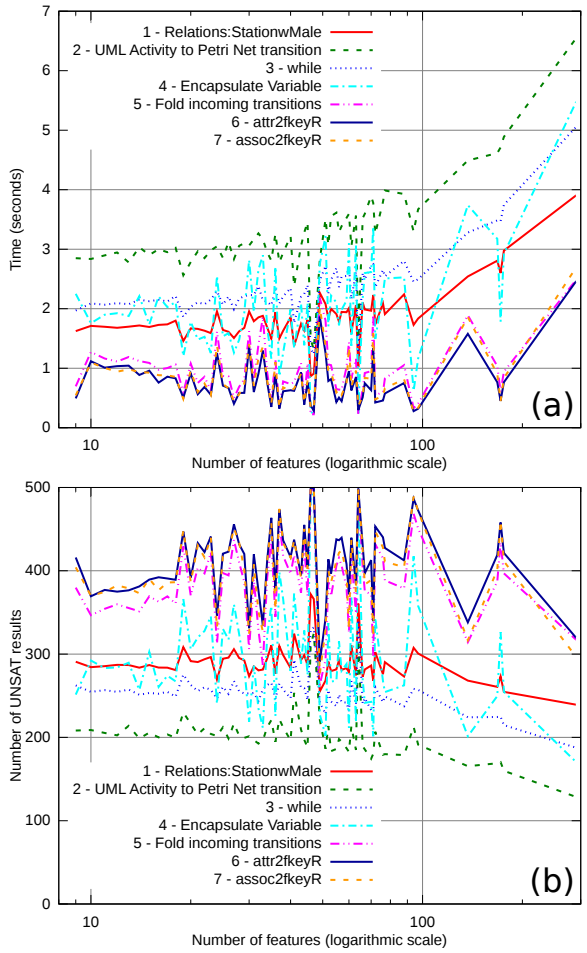

Figure 9: Experimental results - as product line size increases: (a) time to check satisfiability of Φ_1 and Φ_2, (b) number of UNSAT calls.

such as existential quantification, making it quite different and substantially more efficient.

Several works focus on making existing software engineering techniques "variability aware" so that they are applicable on the level of the whole product line rather than individual products – see [44] for a survey. Examples include model checking [9], type checking [26] and testing [27]. Our approach shares the goal of lifting operations to the product line-level, but focuses specifically on model transformations.

Numerous product line-level model transformations allow to derive individual products from a product line [12, 21, 22], merge products and feature models [1, 10, 35], refine feature models [14] and more. Borba et al. [6] organize these works, formally defining a theory of product line refinements as well as a catalog of commonly used refinements. Our work differs from these as we focus on lifting existing transformations from the product to a product line-level rather then handcrafting transformations for product-line specific purposes.

Schulze et al. [38] propose a variant-preserving refactoring approach for feature-oriented product lines [2], aimed to improve the structure of source code. The authors show how to extend traditional approaches to product lines created using feature-oriented programming. Instead, we focus on *annotative* product line representations *realized with models*, and is not limited to just structural improvement.

Several approaches consider the problem of managing the variability of a model transformation itself. Sijtema [41] proposes a method for using a feature model to configure the decisions made by a transformation as it converts an input model into an output model. Kavimandan et al. [28] promote reuse of model transformations through parameterization and specialization of transformation rules. We do not focus on variability of the transformation but rather on applying transformations to assets that contain variability.

Product line evolution approaches, e.g., [32] and [39], focus on studying and supporting scenarios such as splitting, merging, adding or removing features and their implementations. Several such approaches are based on providing templates of "safe" evolution which are to be applied manually. Our focus is rather on transformations that preserve the original set of features, while modifying the structure and the abstraction level of their implementations, and we do so automatically.

Freeman et al. [20] describe an example of "lifting" selected features and their compositions from a product line with "complex" implementations to a product line with "simpler" ones. This work relies on operators mapping higher-level features and their compositions to their lower-level counterparts. Despite a shared name, our work is different: we "lift" transformations rather then product lines.

8. CONCLUSION

MDE and SPLE are key techniques used in modern large scale software development practice. Yet, using these techniques together can pose significant challenges. In particular, classical model transformations designed for use with individual models cannot be reused with product lines of models without substantial modification. In this paper, we addressed this challenge by proposing an algorithm and an accompanying tool for automatically lifting classical model transformations (expressed as graph transformations) to corresponding transformations of product lines. This allows transformations to be reused with no additional development effort, and maintains a clear separation of concerns between the transformation definition and variability management. The initial experiments with the technique showed that it scales well. We believe that transformation lifting is a foundational technique required to address the MDE/SPLE integration problem and we hope that it will help improve the practice of complex software development.

In the future, we plan to do more extensive evaluation of the lifting technique. In addition, we intend to extend it in several ways. (1) Currently, rules are assumed to be executable independently and in any order. Rule control flow mechanisms restrict such applications, and we are interested in lifting rules which are subject to such mechanisms. (2) We are interested in lifting transformations written in other, more programmer-oriented languages, such as ATL [24]. (3) We plan to explore ways to integrate transformation lifting with existing product line tools.

9. ACKNOWLEDGEMENTS

We thank Marcilio Mendonca for his help with the S.P.L.O.T benchmarks. This research has been supported by NECSIS and NSERC.

10. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Comparing Approaches to Implement Feature Model Composition. In *Proc. of ECMFA'10*, pages 3–19, 2010.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

[3] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for in-Place EMF Model Transformations. In *Proc. of MODELS'10*, pages 121–135, 2010.

[4] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *Proc. of SLE'11*, volume 6563 of *LNCS*, pages 296–305. 2011.

[5] E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *Proc. of MODELS'08*, volume 5301 of *LNCS*, pages 53–67. 2008.

[6] P. Borba, L. Teixeira, and R. Gheyi. A Theory of Software Product Line Refinement. *TCS*, 455:2–30, 2012.

[7] A. Capozucca, B. Cheng, G. Georg, N. Guelfi, P. Istoan, and G. Mussbacher. Requirements Definition Document For A Software Product Line Of Car Crash Management Systems. In ReMoDD repository, at `http://www.cs.colostate.edu/remodd/v1/content/bcms-requirements-definition`, 2011.

[8] A. Capozucca, B. H. Cheng, N. Guelfi, and P. Istoan. OO-SPL Modelling of the Focused Case Study. In ReMoDD repository, at `http://www.cs.colostate.edu/remodd/v1/content/bcms-case-study-models-oo-spl-approach`.

[9] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. of ICSE'10*, pages 335–344, 2010.

[10] A. Classen, P. Heymans, T. T. Tun, and B. Nuseibeh. Towards Safer Composition. In *ICSE'09 Companion*, pages 227–230, 2009.

[11] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Ser. in SE. Addison-Wesley, 2001.

[12] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. of GPCE'05*, pages 422–437, 2005.

[13] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems J.*, 45(3):621–645, 2006.

[14] K. Czarnecki, S. Helsen, and U. Eisenecher. Staged Configuration Using Feature Models. In *Proc. of SPLC'04*, pages 266–283, 2004.

[15] K. Czarnecki and A. Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proc. of SPLC'07*, pages 23–34, 2007.

[16] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[17] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[18] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. *CoRR*, cs.DS/9911003, 1999.

[19] M. Famelis, R. Salay, A. Di Sandro, and M. Chechik. Transformation of Models Containing Uncertainty. In *Proc. of MODELS'13*, pages 673–689, 2013.

[20] G. Freeman, D. Batory, G. Lavender, and J. N. Sarvela. Lifting Transformational Models of Product Lines: a Case Study. *J. Software & Systems Modeling*, 9(3):359–373, 2010.

[21] K. Garcés, C. Parra, H. Arboleda, A. Yie, and R. Casallas. Variability Management in a Model-Driven Software Product Line. *Revista Avances en Sistemas e Informática*, 4(2):3–12, 2007.

[22] Ø. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proc. of SPLC'08*, pages 139–148, 2008.

[23] Ø. Haugen, A. Wąsowski, and K. Czarnecki. CVL: common variability language. In *Proc.of the SPLC'12*, pages 266–267. ACM, 2012.

[24] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[25] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of GPCE'08*, pages 35–40, 2008.

[26] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM TOSEM*, 21(3):14, 2012.

[27] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *Proc. of FOSD'12*, pages 1–8, 2012.

[28] A. Kavimandan, A. Gokhale, G. Karsai, and J. Gray. Managing the Quality of Software Product Line Architectures through Reusable Model Transformations. In *Proc. of QoSA/ISARCS'11*, pages 13–22, 2011.

[29] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. of ESEC/FSE'13*, pages 81–91, 2013.

[30] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. of OOPSLA'09*, pages 761–762, 2009.

[31] T. Mens. On the Use of Graph Transformations for Model Refactoring. In *Proc. of GTTSE'05*, volume 4143 of *LNCS*, pages 219–257. 2006.

[32] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the Safe Evolution of Software Product Lines. *ACM SIGPLAN Notices*, 47(3):33–42, 2011.

[33] Object Management Group. *UML Superstructure Specification Version 2.3*, 2010.

[34] D. Plump. Confluence of Graph Transformation Revisited. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, volume 3838 of *LNCS*, pages 280–308. 2005.

[35] J. Rubin and M. Chechik. Combining Related

Products into Product Lines. In *Proc. of FASE'12*, volume 7212 of *LNCS*, pages 285–300. 2012.

[36] J. Rubin and M. Chechik. Quality of Merge-Refactorings for Product Lines. In *Proc. of FASE'13*, pages 83–98, 2013.

[37] R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. McCormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn. An Eclipse-Based Tool Framework for Software Model Management. In *Proc. of Eclipse'07*, pages 55–59, 2007.

[38] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *Proc. of VAMOS'12*, pages 73–81, 2012.

[39] C. Seidl, F. Heidenreich, and U. Aßmann. Co-Evolution of Models and Feature Mapping in Software Product Lines. In *Proc. of SPLC'12*, pages 76–85, 2012.

[40] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[41] M. Sijtema. Introducing Variability Rules in ATL for Managing Variability in MDE-Based Product Lines. *Proc. of MtATL'10*, pages 39–49, 2010.

[42] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel. Refactoring UML Models. In *Proc. of UML'01*, pages 134–148. 2001.

[43] M. Svahnberg, J. Van Gurp, and J. Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[44] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. *School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012*, 2012.

[45] I. Trickovié. Formalizing Activity Diagram of UML by Petri Nets. *Novi Sad J. Math*, 30(3):161–171, 2000.

[46] D. Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In *Proc. of ICGT'02*, pages 378–392, 2002.

[47] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In *Proc. of ICGT'06*, pages 260–274, 2006.