A Variability-Based Approach to Reusable and Efficient Model Transformations

Daniel Strüber[1], Julia Rubin[2], Marsha Chechik[3], and Gabriele Taentzer[1]

[1] Philipps-Universität Marburg, Germany
[2] Massachusetts Institute of Technology, USA
[3] University of Toronto, Canada
{strueber,taentzer}@mathematik.uni-marburg.de,
mjulia@csail.mit.edu, chechik@cs.toronto.edu

Abstract. Large model transformation systems often contain transformation rules that are substantially similar to each other, causing performance bottlenecks for systems in which rules are applied nondeterministically, as long as one of them is applicable. We tackle this problem by introducing *variability-based graph transformations*. We formally define variability-based rules and contribute a novel match-finding algorithm for applying them. We prove correctness of our approach by showing its equivalence to the classic one of applying the rules individually, and demonstrate the achieved performance speed-up on a realistic transformation scenario.

1 Introduction

Model-driven development emerged as a means to combat complexity of large-scale software development through the use of abstraction and refinement. Model-to-model and model-to-code transformations are key enablers of this development paradigm. While there have been many advances in understanding the formal properties of model transformations and devising their development environments, research on maintainability is still in preliminary stages [1]. Large model transformation systems often contain transformation rules that are substantially similar to each other. The most frequently applied mechanism for creating such rules is copying and modifying existing variants. This presents a maintainability obstacle (e.g., all related rules must be updated when a bug is found). The maintainability concern is often combined with a performance concern: In model-driven architecture [2], models go through a series of transformations such as optimizations and code generation, each introducing computational effort.

Inspired by product line engineering approaches [3, 4], a number of existing works, e.g., [5–7] tackle the reuse problem by introducing variability in model transformation rules. These works focus on representing a set of similar rules in a compact manner, providing the user with the ability to later *configure* the rules and produce specific variants. Rule variants are then matched and applied individually, using the classic approach. Since the number of desired configurations of each rule depends on the transformation input which may not be known upfront, the number of configured variants might be high. Thus, even though these works address the maintainability concern by providing a more compact representation of rule sets, they do not offer any performance-related benefits: all variants of a rule must still be considered by the transformation engine.

A. Egyed and I. Schaefer (Eds.): FASE 2015, LNCS 9033, pp. 283–298, 2015.
DOI: 10.1007/978-3-662-46675-9_19

In this paper, we instead propose to augment the transformation engine itself by making it *variability-based*. We handle a scenario where *all* transformation rules need to be considered as long as one of them is applicable. Such an approach is useful in model refactoring suites or translators transforming models between a specific source and target languages. We introduce a novel algorithm for resolving variability *automatically* during the rule matching process, i.e., determination of application sites in the input model. Our central idea is to find matches for the common parts of all rule variants first and then to use them as starting points for the matching of the variable parts. We show that the transformation output produced by our algorithm is equivalent to the one produced when configuring and matching the rules individually, while our approach offers a substantial improvement in performance.

We present our approach to variability-based transformation using graph transformations [8], and, specifically, make the following contributions: (1) a formalization of variability-based rules, investigating their syntax and application semantics on the basis of graph transformation and proving their equivalence to the application of the corresponding classic rules; (2) a novel match-finding algorithm achieving a performance gain when compared to matching the rules individually; (3) an implementation of variability-based model transformation on top of Henshin, a rule-based model transformation language and tool; (4) an evaluation based on a real-life model transformation system that gives evidence of that performance gain.

The remainder of this paper is structured as follows: We introduce a motivating scenario in Sec. 2. In Sec. 3, we give the necessary background and, in Sec. 4, formally define the concept of variability-based graph transformation. We describe the algorithm for directly applying variability-based transformations in Sec. 5 and its implementation in Henshin in Sec. 6. Is effectiveness for model transformations when compared to manipulating a corresponding set of classic model transformation rules is evaluated in Sec. 7. In Sec. 8, we compare our approach with related work. We conclude in Sec. 9 with the summary and discussion of possible future directions.

2 Motivating Example

In this section, we give an example of variability-based transformation rules and their application. Our example is inspired by a set of real-life rules for optimizing and simplifying first-order logic expressions [9], aimed to improve performance of engines that process the expressions, e.g., theorem provers or SAT solvers.

Fig. 1 shows four transformation rules that simplify first-order logic formulas by removing redundant *not* symbols and thus reducing the "depth" of a formula. We present the rules in an integrated form, with the left- and right-hand sides of the transformation being represented in one graph. The elements of this graph have three kinds of labels: *delete*, *preserve*, and *create*. Elements labeled with *delete* and *preserve* are matched to an input model. The former are removed while the latter are kept in the output. Elements labeled with *create* just specify additions to the output.

For the example in Fig. 1, Rule A removes a $\neg\forall\neg$ segment of a formula and transforms it into an $\exists$ segment. This is done by removing nodes #2, #4 and their corresponding edges, replacing the quantifier of node #3 to be "exists" (node #7) rather than "forall" (node #8), and connecting the modified quantifier to the enclosing and the

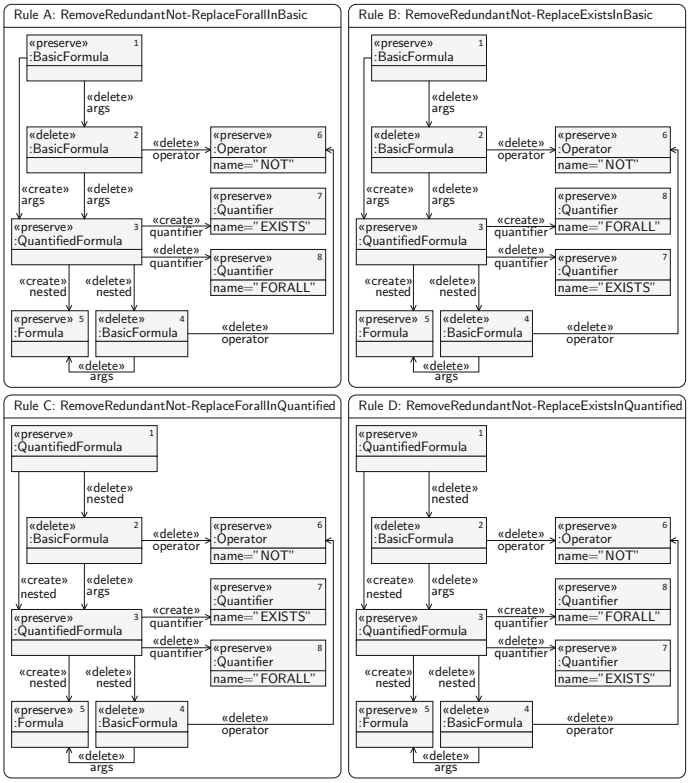

Fig. 1. Four variants of the *Remove Double Negation* refactoring rules

enclosed formulas – nodes #1 and #5, respectively. Similarly, Rule B removes a $\neg\exists\neg$ segment and transforms it into a $\forall$ segment. Rules C and D differ from A and B in the type and adjacent edges of the topmost enclosing formula (element #1): basic vs. quantified. A BasicFormula has an operator and a set of argument formulas, whereas a QuantifiedFormula has a quantifier and nests exactly one other formula. Note that there exists a third kind of formula, PredicateFormula, that encloses no other formulas.

Fig. 2 shows a first-order logic formula $\phi = (\neg\forall x \cdot \neg F(x)) \wedge \mathit{true}$ that can be simplified using one of these rules, namely, Rule A. The formula is also represented as a graph, with formula-specific elements depicted on the left-hand side of the figure. The right-hand side presents a library of "generic" reusable first-order logic operators. Elements #1-#5, #9, #11, #10 match with the corresponding elements #1-#8 of Rule A. We call this assignment a match m_A. Finding m_A triggers the application of

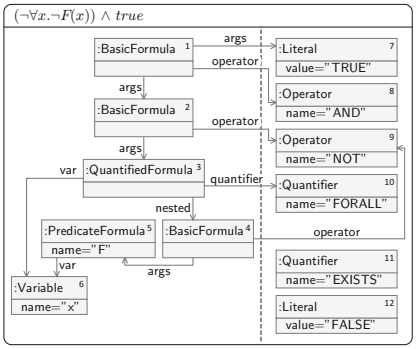

Fig. 2. Example first-order logic formula ϕ

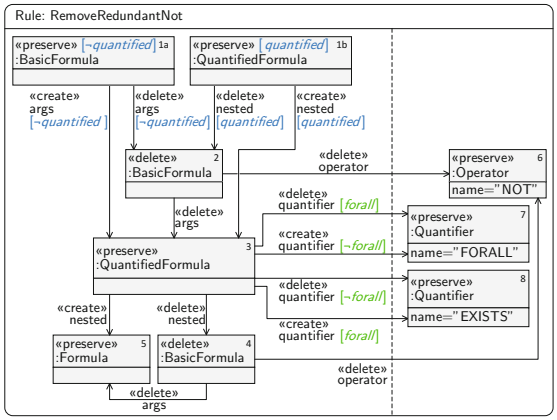

Fig. 3. Variability-based *Remove Double Negation* refactoring rule

Rule A, producing the formula $(\exists x \cdot F(x)) \ \wedge \ \mathit{true}$. Note that m_A is a valid match because PredicateFormula (node #5 in Fig. 2) is a sub-type of the type Formula.

The four rules in Fig. 1 have a lot of commonalities: significant parts of their internal structure and typing are the same. Matching each of these rules with the formula introduces unnecessary complexity and may result in a performance overhead. Fig. 3 shows a compact *variability-based rule* that represents all four individual rules in Fig. 1. The differences between the classic rules are explicitly captured and represented by *variation points*. Rule elements are then *annotated* with *presence conditions* – boolean formulas over the variation points. In the visual representation, annotations are appended in square brackets to the names of their corresponding nodes and edges. For the simplicity of presentation, we omit the presence condition *true*, e.g., for nodes #2-#8.

In our example, there are two variation points: (1) The *forall* variation point controls the direction of the quantifier inversion. When set to *true*, it corresponds to the $\neg\forall\neg$ to $\exists$ inversion, as in rules A and C; when set to *false*, it corresponds to the $\neg\exists\neg$ to $\forall$ inversion, as in B and D. (2) The *quantified* variation point controls the enclosing formula and its adjacent edges. When set to *true*, it corresponds to a formula of the type QuantifiedFormula with outgoing nested edges, as in C and D; when set to *false*, it corresponds to a formula of the type BasicFormula with outgoing args edges, as in rules A and B. Note that this variation pointed cannot be captured using node sub-typing, as it affects edges with different types.

A variability-based rule can be *configured* by setting variation point values and then selecting all elements whose presence conditions evaluate to *true* while removing those whose presence conditions evaluate to *false*. In our example, configuring the rule with *forall=true* and *quantified=false* produces Rule A in Fig. 1 while the configuration *forall=false* and *quantified=true* produces Rule D.

Conceptually, a variability-based rule is equivalent to a set of rules for all its valid configurations. However, the match-finding algorithm for a variability-based rule proposed in this paper performs matching of all its valid configurations at once, thus positively affecting both the maintainability and the performance of the transformation system. The algorithm *automatically* detects a configuration that induces a valid match

using a two-step process. In the first step, it matches the *base rule* – the portion of the rule annotated with *true* and representing common parts of all individual rules. For the example in Figs. 3 and 2, this results in exactly one match, m_{base}, assigning elements #2, #6, #3, #8, #7, #4, #5 to #2, #9, #3, #11, #10, #4, #5 and connecting edges accordingly. In the second step, to match the variable parts the algorithm enumerates the valid configurations (in our example, Rules A to D) and tries to match them using m_{base}. This yields exactly one match for Rule A: m_A. The result of match-finding is m_A paired with the configuration *forall=true* and *quantified=false* that enabled this match.

3 Background: Algebraic Graph Transformation

We present our fundamental approach to variability-based transformation using graph transformation [8]. Graphs can be used to represent the underlying structure of visual models, and their conformance to a metamodel can be formally represented by typed attributed graphs mapped to type graphs. For simplicity, our treatment here uses basic graphs without types, attributes, and constraints, but our implementation and evaluation use the full power of typed attributed graphs, with inheritance, etc. since the concept of variability-basedness is orthogonal to these features. A *directed multi-graph*, simply called a *graph* in the following, comprises a set of nodes and a set of edges connecting these nodes. Structure-compatible mappings between graphs can be expressed in terms of *graph morphisms* which are compatible to source and target functions.

Definition 1 (Graph). *A* graph $G = (G_N, G_E, src_G, trg_G)$ *consists of a set* G_N *of nodes, a set* G_E *of edges, and source and target functions,* $src_G, trg_G : G_E \to G_N$.

Definition 2 (Total (Partial) graph morphism). *Given two graphs* G *and* H*, a pair of total (partial) functions* (f_N, f_E) *with* $f_N : G_N \to H_N$ *and* $f_E : G_E \to H_E$ *forms a total (partial) graph morphism* $f : G \to H$*, a.k.a.* morphism*, if it fulfills the following properties: (1)* $f_N \circ src_G = src_H \circ f_E$ *and (2)* $f_N \circ trg_G = trg_H \circ f_E$*. If both functions* f_N *and* f_E *are injective,* f *is called* injective*. If both functions* f_N *and* f_E *are inclusions,* f *is called* inclusion*.*

In the following, we recall the main definitions of the algebraic approach to graph transformation called the *gluing approach*. In this rule-based approach, graph elements occurring in the left and right-hand sides of a rule, i.e., in an *interface graph*, are used to glue new elements to already existing ones.

Definition 3 (Rule). *A (production) rule* $p = L \xleftarrow{l} I \xrightarrow{r} R$ *consists of graphs L, I and R, called* left-hand side*,* interface graph *and* right-hand side*, respectively, and two injective graph morphisms,* l *and* r*.*

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. Afterwards, unique copies of $R - r(I)$ are added. This behavior can be characterized by a double-pushout [8]. Given a rule and a match, the resulting rule application is unique [8].

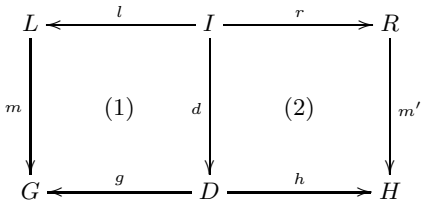

Fig. 4. Rule application by a double pushout (DPO)

Definition 4 (Rule application). *Let a rule $p = L \xleftarrow{l} I \xrightarrow{r} R$ and a graph G with a total graph morphism $m : L \to G$ be given. A* rule application *from G to a graph H, written $G \Rightarrow_{p,m} H$, is given by the diagram in Fig. 4 where (1) and (2) are pushouts. We refer to G, m and H as a* start graph*, a* match*, and a* result graph*, respectively.*

For example, the upper part of Fig. 2 shows a typed attributed graph which can be transformed by applying Rule A from Fig. 1. The rule match w.r.t. nodes has been described in Sec. 2. In addition, the match can be extended to edges. By rule application, nodes #2 and #4 are deleted, together with their adjacent edges. The edge between nodes #3 and #10 is also deleted. As no dangling edges are left behind, the gluing condition is satisfied. Edges between #3 and #9, #1 and #3, as well as between #3 and #4 are created yielding the graph structure for the formula $\phi' = (\exists x \cdot F(x)) \wedge \mathit{true}$.

4 Variability-Based Graph Transformation

In this section, we introduce variability-based graphs and transformation rules and show how to apply them. We provide proofs to all lemmas, propositions, and theorems in this section in an accompanying technical report [10].

4.1 Variability-Based Graphs and Rules

We denote variability using *presence conditions* – propositional expressions over a set of independent *variation points*. The set of these, called a *language of presence conditions*, is fixed for the set of rules and not changed by transformation steps.

Definition 5 (Language of presence conditions). *Given a set of variation points V, $\mathcal{L}_V$ is the set of all propositional expressions over V, called* presence conditions*. A total function $cfg : V \to \{\mathit{true}, \mathit{false}\}$ is a* variability configuration*. cfg* satisfies *a presence condition pc if pc evaluates to true when each variable v in pc is substituted by $cfg(v)$. A presence condition is* valid *if there is a variability configuration satisfying it. A presence condition X is* stronger *than Y iff $X \implies Y$.*

In the example in Sec. 2, $V = \{\mathit{forall}, \mathit{quantified}\}$. *true*, $\neg$*quantified*, and *forall* $\wedge$ *quantified* are valid presence conditions; *forall* $\wedge \neg$*forall* is not valid.

Definition 6 (Variability-based graph). *Given a language of presence conditions $\mathcal{L}_V$, a* variability-based graph *$\hat{G}$ over $\mathcal{L}_V$ is a graph $G = (G_N, G_E, src_G, trg_G)$ and a pair of functions (pc_{G_N}, pc_{G_E}) with $pc_{G_N} : G_N \to \mathcal{L}_V$ and $pc_{G_E} : G_E \to \mathcal{L}_V$ such*

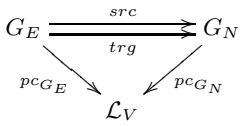

Fig. 5. Variability-based graph

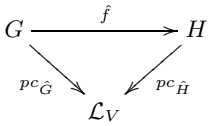

Fig. 6. Variability-based graph morphism

that (1) $\forall e \in G_E \cdot (pc_{G_E}(e) \implies pc_{G_N}(src_G(e))$ *and (2)* $\forall e \in G_E \cdot (pc_{G_E}(e) \implies pc_{G_N}(trg_G(e)))$ *(see Fig. 5). For brevity, we conflate* pc_{G_N} *and* pc_{G_E} *into a single function* $pc_G : (G_N \cup G_E) \to \mathcal{L}_V$ *assuming that* $G_N \cap G_E = \emptyset$*.*

This definition ensures that the presence condition of each edge is stronger than or equal to the presence conditions of both its source and target nodes. Note that pc_{G_N} and pc_{G_E} are total functions, i.e., all graph elements are annotated with presence conditions. Elements which are always present are annotated with *true*. Thus, any graph G without variability can be considered variability-based by defining $\forall x \in G \cdot pc_G(x) := \textit{true}$.

For example, the left-hand side of the variability-based rule in Fig. 3, i.e., all preserved or deleted graph elements, forms a variability-based graph. All graph elements without annotation are mapped to the presence condition *true*, while nodes #1a and #1b and the adjacent edges as well as edges outgoing from node #3 are mapped to the depicted presence conditions.

In the following, we ensure that morphisms and rules over variability-based graphs preserve existing presence conditions.

Definition 7 (Variability-based graph morphism). *Given two variability-based graphs* $\hat{G}$ *and* $\hat{H}$ *over* $\mathcal{L}_V$ *as well as a graph morphism* $\hat{f} : G \to H$*,* $\hat{f}$ *is a* variability-based graph morphism *if* $pc_H \circ \hat{f} = pc_G$ *(see Fig. 6).*

Lemma 1 (Category of variability-based graphs). *Given a fixed* $\mathcal{L}_V$*, variability-based graphs and graph morphisms over* $\mathcal{L}_V$ *form a category.*

Definition 8 (Variability-based rule). *Given* $\mathcal{L}_V$*, a* variability-based rule $\hat{p} = \hat{L} \xleftarrow{\hat{l}} \hat{I} \xrightarrow{\hat{r}} \hat{R}$ *over* $\mathcal{L}_V$ *consists of a span of two variability-based graph morphisms* $\hat{l}$ *and* $\hat{r}$ *over* $\mathcal{L}_V$*. The underlying rule of* $\hat{p}$ *is* $p = (L \xleftarrow{l} I \xrightarrow{r} R)$*.*

For example, Fig. 3 shows a variability-based rule where all preserved graph elements do not change their presence conditions.

4.2 Application of Variability-Based Rules

We now show how to apply variability-based rules: (1) either by flattening them to a set of classic rules and applying a maximal among them in the classic way, or (2) directly, using a suitable variability configuration to identify the corresponding match. We then prove the equivalence of these two approaches.

Variability-Based Transformation through Flattening. We begin by showing how a variability-based rule can be *flattened*, i.e., represented by a set of classic rules.

Definition 9 (Flattening of variability-based graph). *Let a variability-based graph $\hat{G}$ over $\mathcal{L}_V$ be given. For each valid presence condition $c \in \mathcal{L}_V$, $G_c = (G_{c_N}, G_{c_E}, src_c, trg_c)$ is the* flattened graph *iff (1) $\forall n \in G_N \cdot n \in G_{c_N}$ if $c \implies pc_{G_N}(n)$; (2) $\forall e \in G_E \cdot e \in G_{c_E}$ if $c \implies pc_{G_E}(e)$; and (3) $src_c = src_G|_{G_{c_E}}$ and $trg_c = trg_G|_{G_{c_E}}$. Flat($\hat{G}$) is the set of all flattened graphs: $\{G_c \mid c \in \mathcal{L}_V \wedge c \text{ is valid}\}$.*

That is, a flattened graph G_c for presence condition c consists of those elements of $\mathcal{L}_V$ which are annotated by presence conditions implied by c. Note that different conditions can yield the same flattened graphs if the same set of used presence conditions is implied. The set of flattened graphs does not contain graphs for presence conditions equal to *false* since no variability configurations satisfy it.

For example, flattening the left-hand side $\hat{L}$ of the rule in Fig. 3 yields a set of graphs containing the left-hand sides $L_{forall \wedge quantified}$, $L_{\neg forall \wedge quantified}$, $L_{forall \wedge \neg quantified}$ and $L_{\neg forall \wedge quantifed}$ of all the rules in Fig. 1 as well as the intersection of all these – the base left-hand side L_{true}. In addition, *Flat*($\hat{L}$) contains four graphs where only one of the variation points is bound. For all other valid presence conditions $pc \in \mathcal{L}_V$, L_{pc} is equal to one from this list.

Lemma 2 (Smallest graph in flattening). *G_{true} is the smallest subgraph of G in Flat($\hat{G}$).*

The flattening of graphs can be lifted to graph morphisms and rules straightforwardly, yielding the rules ordered by the implication of their presence conditions to ensure that application of larger rules, modeling more specific cases, is attempted first.

Definition 10 (Flattening of variability-based graph morphism). *Let a variability-based graph morphism $\hat{f} : \hat{G} \to \hat{H}$ be given.* Flattening of *$\hat{f}$ is Flat($\hat{f}$) = $\{f_c : G_c \to H_c \mid c \in \mathcal{L}_V \wedge c \text{ is valid}\}$ with $G_c \in$ Flat($\hat{G}$), $H_c \in$ Flat($\hat{H}$) and $f_c = f|_{G_c}$.*

Definition 11 (Flattening of variability-based rule). *Given a variability-based rule $\hat{p} = \hat{L} \xleftarrow{l} \hat{I} \xrightarrow{r} \hat{R}$ over $\mathcal{L}_V$, we can apply the flattening of morphisms twice: Flat($\hat{p}$) = $(\{p_c : L_c \xleftarrow{l_c} I_c \xrightarrow{r_c} R_c \mid c \in \mathcal{L}_V \wedge c \text{ is valid}\}, \leq)$ with $l_c : I_c \to L_c \in$ Flat($\hat{l}$), $r_c : I_c \to R_c \in$ Flat($\hat{r}$). For the resulting rule set, a partial order between rules is defined through implication between their presence conditions: $p_{c_1} \leq p_{c_2}$ iff $(c_2 \implies c_1)$. Rule $p_{true} \in$ Flat($\hat{p}$) is also called* base rule.

For example, flattening the rule in Fig. 3 yields a set containing the four rules shown in Fig. 1 as well as their common maximal sub-rule (being the base rule) – the rule in Fig. 3 with only elements annotated by *true*.

The base rule is smaller than all the other rules in the set w.r.t. the partial order $\leq$. All rules of Fig. 1 are incomparable to each other. The additional four rules are larger than the base rule but smaller than the rules in Fig. 1.

Definition 12 (Ordered rule set). *An ordered rule set $\mathcal{R} = (\mathcal{R}_{rules}, \leq)$ consists of a set $\mathcal{R}_{rules}$ of rules and a partial order $\leq$ over this set.*

Definition 13 (Application of an ordered rule set). *Given an ordered rule set $\mathcal{R}$ and a graph G, the application of $\mathcal{R}$ to G is the set of rule applications: Trans($\mathcal{R}, G$) =*

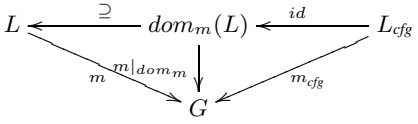

Fig. 7. A match induced by variability configuration

$\{G \Rightarrow_{p,m} H\}$ *with* $p \in \mathcal{R}_{rules}, p = (L \xleftarrow{l} I \xrightarrow{r} R)$ *and a match* $m : L \to G$ *and* $\forall p' \in \mathcal{R}_{rules}$ *with* $p' \geq p : \neg\exists$ *a match* $m' : L' \to G$ *with* $m'(L') \supset m(L)$.

For example, for the graph of formula ϕ in Fig. 2, there is exactly one match of base rule p_{true}. However, this rule is not maximal – Rule A = $p_{forall \wedge \neg quantified}$ in Fig. 1 can be matched as well. This match includes the match of the base rule, i.e., it is larger, and there is no larger one. For the graph structure of the formula $((\neg\forall x \cdot \neg F(x)) \wedge \mathit{true}) \wedge (\neg\forall x \cdot \neg F(x)) \wedge \mathit{true})$, Rule A can be applied twice and there are no larger rules that match.

Direct Application of Variability-Based Rules. In the following, we consider the direct application of variability-based rules by finding a suitable variability-based match on-the-fly. The central task is to find a variability configuration such that the part of the left-hand side that can be matched is *locally maximal*, i.e., the match of a rule part cannot be extended by variable parts. If the resulting partial morphism of the left-hand side to graph G satisfies the gluing condition for the corresponding flat rule, the rule application can take place.

Definition 14 (Maximal partial morphism). *Given two graphs* G *and* H*, let* $PM_{G,H}$ *be the set of all partial graph morphisms from* G *to* H*. A partial morphism* $m \in PM$ *is* maximal *if* $\forall m' \in PM \cdot \neg(dom_{m'}(G) \supset dom_m(G))$.

Definition 15 (Variability-based match). *Given a variability-based rule* $\hat{p}$ *over* $\mathcal{L}_V$ *and a graph* G*, a* variability-based match $\hat{m} = (m, cfg)$ *over* $\mathcal{L}_V$ *consists of a maximal partial morphism* $m \in PM_{L,G}$ *and a variability configuration* $cfg : V \to \{true, false\}$ *such that* $\forall x \in dom_m(L) \cdot cfg$ *satisfies* $pc_L(x)$*. cfg induces a rule* p_{cfg} *s.t. cfg satisfies all presence conditions occurring in* p_{cfg}*. Moreover, reducing* m *to its domain, we get a morphism* m_{cfg} *which has to satisfy the gluing condition w.r.t.* p_{cfg} *(see Fig. 7).*

To apply the rule in Fig. 3 to the graph for formula ϕ in Fig. 2 by mapping to the same elements as Rule A in Fig.1, we choose the variability configurations *cfg(quantified)* = *false* and *cfg(forall)* = *true*. Thus, p_{cfg} is Rule A. The resulting morphism $m_{cfg} = m_A$ satisfies the gluing condition, hence, it is a match for Rule A.

In the following, we show that the matched left-hand side of the variability-based rule is exactly the left-hand side of the chosen flat rule and there is no larger rule whose match would comprise the chosen one.

Proposition 1 (Variability-induced rule). *Given a variability-based rule* $\hat{p}$ *with a variability-based match* $\hat{m} = (m, cfg)$ *to graph* G*,* $\hat{m}$ *induces a rule* p_{cfg} *with the following properties: (1)* $p_{cfg} \in Flat(\hat{p})$*; (2)* $L_{cfg} = dom_m(L)$*, and (3)* $\neg\exists p' \in Flat(\hat{p})$ *s.t.* $p_{cfg} \leq p'$ *and cfg satisfies* $pc_{L'}(x)$*,* $\forall x \in L'$.

Definition 16 (Application of a variability-based rule). *Given a match $\hat{m}$ for variability-based rule $\hat{p}$ and graph G, the* application of $\hat{p}$ at $\hat{m}$ *is the classic rule application of p_{cfg} to m_{cfg} induced by $\hat{m}$ leading to rule application $G \Rightarrow_{p_{cfg},m_{cfg}} H$.*

Applying the rule in Fig. 3 to the graph of formula ϕ in Fig. 2 at the variability-based match computed in the example after Def. 15 yields the graph structure of formula ϕ' described at the end of Sec. 3. Now, we show that the set of all applications of a variability-based rule $\hat{p}$ to a graph G is equal to the set of classic rule applications obtained from flattening $\hat{p}$ and applying these rules to G.

Theorem 1 (Equivalence of rule applications). *Given a variability-based rule $\hat{p}$ and a graph G, the following holds: $\{G \Rightarrow_{\hat{p},\hat{m}} H |\ \hat{m} = (m, cfg)$ with $m \in PM_{L,G}\} = Trans(Flat(\hat{p}), G)$.*

5 Variability-Based Matching Algorithm

In this section, we describe an algorithm for implementing the concept of variability-based match (Def. 15). Our guiding intuition is to find matches for the base rule first, then expand these matches for the variable parts and finally filter the result to contain only maximal mappings.

Matching the base rule (see Def. 11) yields matches for the common parts that we store in a collection called *baseMatches*. Function FINDMATCHES in Fig. 8 extends *baseMatches* to find matches for the variable parts. It enumerates all consistent variability configurations, derives the corresponding rules and matches them classically. FINDMATCHES receives an input model, a variability-based rule, the *baseMatches* set, and two intermediate parameters: a data structure *bindings* that assigns each of the rule's presence conditions to one of the literals *true*, *false* or *unbound* (initially all entries are set to *unbound*) and a set to accumulate variability-based matches (initially empty). The function outputs the set of variability-based matches.

An execution of FINDMATCHES systematically binds all presence conditions, starting on Line 2 with an arbitrary one that we call pc_0. To enumerate all valid configurations, we first set pc_0 to *true* and then to *false* (Lines 3-4 and 5-6). In both calls to FINDMATCHESINNER, we first consider those presence conditions that were previously unbound and now are either contradicting or implied by the current bindings. On Lines 10 and 11, we compute them using a SAT solver, calling the results $bindings_{\not\downarrow}$ and $bindings_{\rightarrow}$ (for *false* elements and *true* elements, respectively). We update the bindings accordingly on Line 12. If all presence conditions are now bound, the problem becomes classic matching. We determine the classic rule to be matched by removing rule elements with a *false* presence condition on Line 14. The classic match-finder tries to bind the rule elements contained in the derived rule, but not in the base rule. The computed matches are translated into variability-based matches, being pairs of a classic match and the current variability configuration, on Lines 15-16. If some presence conditions have not been bound, we call FINDMATCHES again on Line 18. On Lines 7 and 19, we reset temporary bindings of variables to clean up before backtracking. To retain only the maximal matches, as demanded by Def. 15, we clean up after the outer FINDMATCHES call by removing all non-maximal entries from the result.

```
Input: model: Input model
Input: rule: Variability-annotated rule
Input: baseMatches: Classic matches of the base rule
Input: bindings: {Presence conditions used in rule} → {true, false, unbound}
Input: matches: Accumulated variability-based matches
Output: matches: Accumulated variability-based matches
 1: function FINDMATCHES(model, rule, baseMatches, bindings, matches)
 2:     pc0 = bindings.select(unbound).get(0)
 3:     bindings.set(pc0, true)
 4:     FINDMATCHESINNER(model, rule, baseMatches, bindings, matches)
 5:     bindings.set(pc0, false)
 6:     FINDMATCHESINNER(model, rule, baseMatches, bindings, matches)
 7:     bindings.set(pc0, unbound)
 8:     return matches
 9: function FINDMATCHESINNER(model, rule, baseMatches, bindings, matches)
10:     bindings↯ = bindings.select(unbound).select(p | bindings.contradicts(p))
11:     bindings→ = bindings.select(unbound).select(p | bindings.implies(p))
12:     bindings.setAll(bindings↯ → false, bindings→ → true)
13:     if bindings.select(unbound).isEmpty() then
14:         classicRule = rule.minus( {x ∈ rule | x.pc ∈ bindings.select(false)} )
15:         classicMatches = Matcher.matchClassically(model, classicRule, baseMatches)
16:         matches.addAll(createVariabilityBasedMatches(classicMatches))
17:     else
18:         FINDMATCHES(model, rule, baseMatches, bindings, matches)
19:     bindings.setAll(bindings↯ → unbound, bindings→ → unbound)
20:     return
```

Fig. 8. Pseudocode for recursive function FINDMATCHES

To exemplify our algorithm, we continue with the scenario at the end of Sec. 2. First, we create and match the base rule, comprising the elements annotated with *true*, by classic match-finding. The computed *baseMatches* set contains exactly match m_{base}. We arbitrarily select a presence condition $\neg$*qualified* and set it to *true* on Line 3, thus deriving *qualified* to be *false* on Lines 10-12. To bind the rest of the presence conditions, we call FINDMATCHES again on Line 18. We then select *forall* and set it to *true*, thus setting $\neg$*forall* to *false* and completing the binding of presence conditions. On Line 14, we remove all rule elements labelled $\neg$*forall* or $\neg$*qualified* to derive Rule A. Calling the classic match finder on this rule on Line 15 yields m_{A}. We pair this classic match with the current bindings to create a variability-based match. The remaining three configurations are determined analogously; however, they do not yield any additional matches.

Complexity of our algorithm is determined by the number of configurations which grows exponentially with the number of variation points. Of course, the configurations determine rules that in the classic approach would be matched individually. Thus, complexity of our algorithm is the same as that in classic matching. Yet, since we save matching effort by precomputing base matches and then extending them, we expect our algorithm to perform better than the classic one. We experimentally compare performance of our approach with classic in Sec. 7.

6 Implementation

Our implementation is based on the Henshin model transformation suite [11] which provides basic transformation functionalities for classic rules. Henshin consists of a transformation meta-model, a graphical editor for rule specification, and an interpreter engine for rule application. To *specify* variability-based rules, we extended the meta-model and editor of the Henshin language, allowing annotations of rule elements with presence conditions in the *properties* view. The user can highlight groups of rule elements sharing the same presence condition by assigning colors. To *apply* variability-based rules, we extended the Henshin interpreter engine, implementing the algorithm described in Sec. 5. We used FeatureExprLib [12], a tool which computes valid configurations of features using a SAT solver, for evaluating presence conditions. Finally, we cached the results of all evaluations in order to avoid repeating the same computations.

Our implementation also allows the user to restrict the set of valid configurations by defining relationships between variation points, such as *mutual exclusion* and *require*. These relationships can also be specified in terms of just presence conditions, e.g., setting a condition to $A \wedge \neg B$ if a variation point A excludes B.

7 Evaluation

In this section, we aim to answer two research questions: (RQ1) How compact are rule sets with variability-based rules compared to classic? (RQ2) What is the speedup of applying rules with variability instead of the corresponding classic ones?

Scenario. We investigated a transformation system comprised of 54 classic transformation rules. The rules constitute a translator from Object Constraint Language (OCL) expressions to nested graph constraints [9].

In this system, the main performance bottleneck, which we call *bottleneck rule subset* (BRS), is a subset of 36 rules that are applied nondeterministically, as long as one of them can be matched. The left-hand sides of the BRS rules have between 9 and 37 graph elements and share a considerable amount of commonalities. We applied the transformation system to 10 constraints described in [9] – an assortment of OCL constraints designed for a large coverage of applicable rules. The size of the input models, comprising individual constraints as well as the OCL standard library, containing operators and literals referenced by the constraints, ranges from 1832 to 1854 model elements.

Setup and Metrics. We manually refactored the 36 classic rules in BRS into 10 variability-based ones, relying on name similarities. We merged the original rules and annotated the result with presence conditions. To ensure correctness of the refactoring, we checked equality of the models yielded by both the original and the variability-based rule sets.

To investigate RQ1, we measured two metrics on both rule sets: *number of rules* and *number of elements per rule*, allowing us to quantify compactness. To investigate RQ2, we measured the *execution time* on both rule sets, allowing us to quantify performance. We determined the execution time on a Windows 7 workstation with a 3.40 GHz Intel i7-3770 processor and 8 GB of RAM.

Results of RQ1. In our example, variability-based rules help decrease the number of rules by 72% while increasing the number of elements per rule by 17%. Specifically, from 36 rules with the total of 1281 nodes and 1764 edges, we extracted 10 variability-based rules with 399 nodes and 589 edges and 2-3 variation points each. The ratio between common and variable parts increased with the size of the rule: the smallest rules had 10 common and 34 variable elements; the median – 69 common and 34 variable elements; the largest – 102 common and 60 variable elements.

Results of RQ2. Table 1 shows the result of applying the classic and the variability-based rule sets on each model, repeating the experiment 10 times.

We show the mean time *(mean)* and standard deviation *(sd)* for each rule set and model. For three of the input models, *ocl01* to *ocl03*, no performance difference was observable. For the remaining seven models, the execution time of transformations using rules with variability was on average 3.9 times faster than with the classic rules. To examine the cause of the performance difference more closely, we counted the number of successful and failed matching attempts (for a detailed account, please refer to [10]). In accordance with Theorem 1, the number of successful rule applications was always the same for both rule sets. In our approach, for *ocl04* to *ocl09* the number of failed match attempts is substantially lower, 1.72 times on average. We explain this observation by our reduced number of rules that increases the ratio of applicable to total ones. Overall, our experiments showed that in a scenario with a considerable amount of variability between rules, our approach allowed to create more compact rules and considerably improve the performance of their application.

Table 1. Running time

	time (sec) classic		*time (sec) var.-based*	
model	mean	sd	mean	sd
ocl01	**<.1**	<.1	**<.1**	<.1
ocl02	**<.1**	<.1	**<.1**	<.1
ocl03	**<.1**	<.1	**<.1**	<.1
ocl04	**56.7**	10.6	**14.2**	4.5
ocl05a	**65.1**	9.2	**13.0**	3.4
ocl05b	**96.7**	20.4	**19.7**	4.8
ocl06	**49.0**	13.4	**11.5**	3.9
ocl07	**389.4**	93.4	**78.4**	3.5
ocl08	**191.0**	11.7	**48.4**	12.7
ocl09	**11.6**	2.6	**5.0**	1.5
average	**85.9**	16.1	**19.0**	3.4

Threats to Validity and Limitations. The most important threat to validity is our choice of transformation rules and input models that may not be representative. We attempted to mitigate it by selecting a set of realistic transformation rules and input models already studied in the literature.

The performance gain achieved by our approach is affected by the amount of variability appearing within the rules. The maximum performance gain is observed for rule bases with large common parts which we match globally, paired with small variable parts which we match individually. Since the ratio of common and variable parts observed in our study may not be the same in all systems, the results might be different. Yet, matching common parts of similar rules only once is still expected to result in performance improvements. Furthermore, we are aware of the following caveats: (1) for very small examples, the overhead of variability processing might outweigh the reduced matching costs; and (2) if the left-hand side of the base rule does not represent a

connected graph and the left-hand sides of the rule variants do, matching the base rule might become more expensive. We intend to investigate this issue in the future.

8 Related Work

The variability-based rules introduced in this paper are inspired by annotative representations of product lines [13–15] and augment representations proposed in earlier works.

While our focus is on the batch processing of *all* valid configurations of a variability-based rule, a number of related approaches, e.g., [5–7], target scenarios where a rule configuration is set externally to derive a desired classic rule. In such cases, [5, 6] report on a trade-off between better variability management and a performance overhead, the latter caused by the derivation of rules. In contrast, variability-based rules and matching improve both the compactness *and* the performance of a transformation system.

As for expressiveness, [5] and [7] are based on creating refinement rules for the variable parts and assigning them to one feature (or variation point). In turn, we support propositional presence conditions over variation points. In our evaluation example, we avoided several redundancies by assigning rule elements to a conjunction of two variation points. In this respect, [6] goes even further by allowing users to annotate a rule element with embedded C++ code, which, however, would produce an extremely large search space for variability-based matching.

Several model transformation languages implement *rule refinement* [1] – an important mechanism for reuse inside the same transformation system. In such languages, a base rule is refined by a set of sub-rules modifying it. Then, some approaches [16, 17] flatten the rules for application, i.e., compile them into simpler rules. The translational semantics in the approach proposed in RubyTL [18] is closest to ours – it applies the base rules first and then applies the refinement rules on the target model of the transformation. In contrast, our approach aims to efficiently find matches in the *source* model.

In [19], the authors propose an approach for transformation "lifting": given a classic model transformation, a transformation that operates on a family of related models is generated automatically. Instead, we do not focus on transforming a family of models but rather on creating and applying a family of related transformation rules in an efficient manner. [20] presents a reuse concept based on abstract transformation rules that can be instantiated for variants of similar meta-models. The abstract transformation rules are reverse engineered from existing transformation rules. In [21], the authors apply incremental graph pattern matching based on Rete networks to improve performance of transformation systems. However, they target the use case of successive application of the same set of rules on a modified input model and do not deal with variability inside the transformation system. These approaches are orthogonal to ours, and we intend to combine them with ours in the future.

9 Conclusion

In this paper, we proposed a novel approach to improve reuse and performance in model transformation systems. Aiming to handle a class of problems where rules with many

commonalities are to be applied nondeterministically as long as one of them is applicable, we introduced variability not only to the rules but also to transformations using them. We proved correctness of our approach and contributed an efficient matching algorithm evaluated using a realistic model transformation system.

In this work, the refactoring of classic to variability-based rules was performed manually. As a future work, we intend to automate this step, possibly by applying techniques proposed by the product line engineering community for determining commonalities and variabilities in models. Moreover, while this work focused on rule application, other computationally expensive operations performed on rules, such as state-space exploration or critical pair analysis, might also benefit from explicit variability management. We intend to investigate this in the future. Providing an efficient solution for the matching of base rules represented as disconnected graphs is also subject for possible future work, as is to compare our approach against existing algorithms aiming at specific tasks in compilers and theorem provers. Finally, we aim to apply variability-based rules to distributed modeling scenarios with multiple variants of editing steps, e.g., synchronous and asynchronous ones [22].

Acknowledgements. We thank Thorsten Arendt and Frank Hermann for providing input for our evaluation.

References

1. Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: Reuse in Model-to-Model Transformation Languages: Are We There Yet? In: SoSyM, pp. 1–36 (2013)
2. Soley, R.: Model Driven Architecture. Object Management Group (2000)
3. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
4. Pohl, K., Boeckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
5. Sijtema, M.: Introducing Bariability Rules in ATL for Managing Variability in MDE-based Product Lines. In: Proc. of MtATL 2010, pp. 39–49 (2010)
6. Kavimandan, A., Gokhale, A., Karsai, G., Gray, J.: Managing the Quality of Software Product Line Architectures through Reusable Model Transformations. In: Proc. of QoSA/ISARCS 2011, pp. 13–22. ACM (2011)
7. Trujillo, S., Zubizarreta, A., De Sosa, J., Mendialdua, X.: On the Refinement of Model-to-Text Transformations. In: Proc. of JISBD 2009, pp. 123–133 (2009)
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories. Fundamenta Informatica 74, 31–61 (2006)
9. Arendt, T., Habel, A., Radke, H., Taentzer, G.: From Core OCL Invariants to Nested Graph Constraints. In: Giese, H., König, B. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 97–112. Springer, Heidelberg (2014)
10. Strüber, D., Rubin, J., Chechik, M., Taentzer, G.: A Variability-Based Approach to Reusable and Efficient Model Transformation - Technical Report, `https://www.uni-marburg.de/fb12/swt/research/publications`

11. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
12. Kenner, A., Kästner, C., Haase, S., Leich, T.: TypeChef: Toward Type Checking #ifdef Variability in C. In: Proc. of FOSD 2010, pp. 25–32 (2010)
13. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
14. Kästner, C., Apel, S.: Integrating Compositional and Annotative Approaches for Product Line Engineering. In: Proc. of the Wksp. on Modularization, Composition and Generative Techniques for PLE (McGPLE) at GPCE 2008, pp. 35–40 (2008)
15. Rubin, J., Chechik, M.: Combining related products into product lines. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering. LNCS, vol. 7212, pp. 285–300. Springer, Heidelberg (2012)
16. Anjorin, A., Saller, K., Lochau, M., Schürr, A.: Modularizing Triple Graph Grammars Using Rule Refinement. In: Gnesi, S., Rensink, A. (eds.) FASE 2014 (ETAPS). LNCS, vol. 8411, pp. 340–354. Springer, Heidelberg (2014)
17. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: Atl: A qvt-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, pp. 719–720. ACM (2006)
18. Cuadrado, J.S., Molina, J.G.: A Model-Based Approach to Families of Embedded Domain-Specific Languages. IEEE TSE 35, 825–840 (2009)
19. Salay, R., Famelis, M., Rubin, J., Sandro, A.D., Chechik, M.: Lifting Model Transformations to Product Lines. In: Proc. of ICSE 2014, pp. 117–128 (2014)
20. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Reverse engineering of model transformations for reusability. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 186–201. Springer, Heidelberg (2014)
21. Bergmann, G., Ráth, I., Szabó, T., Torrini, P., Varró, D.: Incremental pattern matching for the efficient computation of transitive closure. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 386–400. Springer, Heidelberg (2012)
22. Strüber, D., Taentzer, G., Jurack, S., Schäfer, T.: Towards a distributed modeling process based on composite models. In: Cortellessa, V., Varró, D. (eds.) FASE 2013 (ETAPS 2013). LNCS, vol. 7793, pp. 6–20. Springer, Heidelberg (2013)