Reusing Model Transformations While Preserving Properties

Ethan K. Jackson[1], Wolfram Schulte[1],
Daniel Balasubramanian[2], and Gabor Karsai[2]

[1] Microsoft Research
{ejackson,schulte}@microsoft.com
[2] Vanderbilt University
{daniel,gabor}@isis.vanderbilt.edu

Abstract. Model transformations are indispensable to *model-based development* (MBD) where they act as translators between *domain-specific languages* (DSLs). As a result, transformations must be verified to ensure they behave as desired. Simultaneously, transformations may be reused as requirements evolve. In this paper we present novel algorithms to determine if a reused transformation preserves the same properties as the original, without expensive re-verification. We define a type of behavioral equivalence, called *lifting equivalence*, relating an original transformation to its reused version. A reused transformation that is equivalent to the original will preserve all compatible universally quantified properties. We describe efficient algorithms for verifying lifting equivalence, which we have implemented in our FORMULA [1, 2] framework.

1 Introduction

Model-based development (MBD) utilizes *domain-specific languages* (DSLs) and *model transformations* to support formal modeling [3–6] . DSLs are used to (1) capture vertical abstraction layers, (2) separately specify design concerns, and (3) provide convenient modeling notations for complex problem domains [7]. Model transformations act as bridges between DSLs in order to (1) incrementally refine models through abstraction layers, (2) compose models into a consistent whole or evolve them as requirements evolve [2], and (3) capture operational semantics as sequences of transformation steps [8].

Consequently, composition, verification, and reuse of DSLs/transformations are essential operations. Informally, a DSL X exposes an abstract syntax $S(X)$, and a model transformation τ is a mapping across syntaxes. A verified transformation is a mapping guaranteed to exhibit certain properties, such as every well-formed input yields a well-formed output. Transformations are reused whenever a new transformation τ' is built from parts of an existing τ. We explore whether the properties of τ also hold in the reused transformation τ'.

For example, consider a DSL for a *non-deterministic finite state automaton* (NFA) abstraction. An important operation on NFAs is the synchronous product $\otimes$, which creates product NFAs where states have internal structure

D.S. Rosenblum and G. Taentzer (Eds.): FASE 2010, LNCS 6013, pp. 44–58, 2010.

(i.e. pairs of product states). Let *ProdNFA* be the DSL for product NFAs, then the $\tau_{\otimes}$ transformation is a mapping from $S(NFA) \times S(NFA)$ to $S(ProdNFA)$. It can be verified that $\tau_{\otimes}$ has the property: $\forall x_1, x_2 \ \#states(\tau_{\otimes}(x_1, x_2)) = \#states(x_1) \times \#states(x_2)$, i.e. products grow combinatorially in size. This convenient transformation can be reused to create products of product NFAs: $\tau'_{\otimes} : S(ProdNFA) \times S(ProdNFA) \rightarrow S(ProdNFA)$. We would like to know if the previous property holds in the new context without reproving it.

In this paper we present a novel approach to avoid expensive re-verification when a model transformation is reused in a new context. Our approach is to fix an auxiliary class of transformations $\mathcal{T}_{rw}$, which we call *rewriting procedures*. Whenever a transformation is reused in a new context, an attempt is made to discover a rewriting procedure from the new to the old context. A reused transformation τ' *lifts* τ if it is equivalent to a rewriting procedure followed by an application of the original transformation τ. If this is the case, then all (compatible) *first-order universally quantified* properties of τ also hold for τ', but with dependencies on the rewriting procedures. Finally, for reasonable choices of $\mathcal{T}_{rw}$, these rewriting procedures can be eliminated from lifted properties resulting in an equivalent property that lives completely within the new context. We have implemented this approach in our FORMULA framework [2]: Static analysis automatically rejects reused transformations that should maintain properties, but for which lifting cannot be verified.

This paper is divided into the following sections: Section 2 describes related work. Section 3 presents a general formal framework. Section 4 explains our implementation of the general framework. We conclude in Section 5.

2 Related Work

Much work on transformation reuse is targeted at the automated or semi-automated evolution of transformations in response to either refactored models or language constructs. For instance, [9] describes how the evolution of models may break transformations intended to operate on these models. The solution is to capture model refactorings as transformations, which can then be used to upgrade transformations that are unaware of these refactorings. [10] deals the evolution of transformation context through user-defined rules relating the constructs in the original meta-models with those in the evolved meta-models. These rules are used to upgrade the transformation as much as possible.

Functional programming solves a related reuse problem: Given a function $f : X \rightarrow Y$, can f be applied to new recursive data structures containing data of type X? For example, if $f : \mathbb{Z} \rightarrow \mathbb{Z}$, determine a function $f' : Lists(\mathbb{Z}) \rightarrow \mathbb{Z}$ from lists of integers to integers that generalizes f. In this case, the lifted f' is not behaviorally equivalent to f, but may preserve properties of f depending on the choice of f'. The Bird-Meertens [11] formalism enumerates patterns of recursive data types that can be used to generalize f automatically. See [12] for a catalog of these recognizable patterns and the formal properties of these generalizations. The main application of this work has been to automatically parallelize functional programs [13].

There is also an important body of work on transformation verification. Verification of model transformations can be performed point-wise, on the input/output pairs of a transformation, or on the transformation itself. The former is known as *instance-based* verification. [14] describes an approach where each execution of the transformation is verified by checking whether the output model *bi-simulates* the input model. [15] uses a set of graph transformation rules to describe the operational semantics of a DSL and then generates a transition system for each well-formed model of the language. A model checker is used to provide formal verification and check dynamic properties of the models.

[16] is an example of verifying the action of a transformation τ over its entire domain. Here, it is assumed that the behaviors of source and target models are defined by simulation rules (which are also transformations). A transformation is correct/complete if for every input model the output model includes the behaviors of the input, and vice versa. This is verified by examining the effects of τ on the simulation rules. Modular verification of model refactorings is also described in [17, 18]. The authors show that once a behavioral semantics is fixed for models of a DSL (e.g. models may correspond to CSPs), then model refactorings can be shown to preserve behavior under certain conditions.

3 General Framework

3.1 Transformations and Reuse Scenarios

We begin with a general discussion of DSLs and transformations. For our purposes, a DSL X is an object providing a set $S(X)$, called the *abstract syntax of* X. A model x is instance of the abstract syntax; equivalently x is an element of $S(X)$. A *model transformation* τ is a map from n input models to m output models:

$$\tau : S(X_1) \times \ldots \times S(X_n) \rightarrow S(Y_1) \times \ldots \times S(Y_m) \tag{1}$$

The domain/range of τ form its context. A transformation is defined via a set of *rules* R, which match patterns in input models to build output models. A transformation *terminates* when no more rules can be applied. Rules are variously specified as graph-rewrite rules [3] [19], declarative relations [20], term-rewrites rules [21], logic programs [2], and even blocks of imperative code [22]. A set of rules R is converted to a mapping τ by a formal semantics $[\![\ ]\!]$, which also takes into account the input/output DSLs:

$$[\![R, \overline{X}, \overline{Y}]\!] \mapsto \tau, \text{ where } \overline{X} = [X_1, \ldots, X_n] \text{ and } \overline{Y} = [Y_1, \ldots, Y_m]. \tag{2}$$

Since rules operate on abstract syntax, the context $(\overline{X}, \overline{Y})$ is required to bind patterns in the rules with elements of the sytnax. We do not hypothesize on the framework-independent properties of $[\![\ ]\!]$, other than to assume that, when defined, τ is a *function* whose signature is given by (1). Later in the paper we investigate $[\![\ ]\!]$ for a particular transformation framework.

Using this notation, we study reuse scenarios where the rules R are interpreted in a new context $(\overline{X'}, \overline{Y'})$:

$$[\![R, \overline{X'}, \overline{Y'}]\!] \mapsto \tau', \text{ where } \overline{X'} = [X'_1, \ldots, X'_n] \text{ and } \overline{Y'} = [Y'_1, \ldots, Y'_m]. \quad (3)$$

We provide static analysis to decide if τ' preserves the properties of τ without expensive re-verification. Several important scenarios are included in this definition:

Example 1. **Transformation Evolution.** A transformation is defined and verified, but changes in requirements necessitate changes in DSL syntax [10]. In this case τ evolves to τ', where each X'_i (or Y'_j) is either the original X_i (or Y_j) or a modified version X^*_i (or Y^*_j).

$$X'_i = \begin{cases} X^*_i \text{ if requirements change} \\ X_i \text{ otherwise} \end{cases}, \quad Y'_j = \begin{cases} Y^*_j \text{ if requirements change} \\ Y_j \text{ otherwise} \end{cases}. \quad (4)$$

Example 2. **DSL Composition.** A DSL X may represent one *aspect* or *architectural facet* of a multi-faceted design problem. In this case, a complete abstraction is formed by composing X with another DSL X^* to obtain $X' = X \oplus X^*$. The DSL composition operator $\oplus$ varies across tools from *UML package merge* to *eBNF grammar composition* [23]. It is then necessary to reuse τ across composite DSLs.

$$X'_i = \begin{cases} X_i \oplus X^*_i \text{ if composed} \\ X_i \text{ otherwise} \end{cases}, \quad Y'_j = \begin{cases} Y_j \oplus Y^*_j \text{ if composed} \\ Y_j \text{ otherwise} \end{cases}. \quad (5)$$

3.2 Properties of Transformations

A *quantifier-free formula over* τ is a well-formed formula consisting of variables, function applications, and τ applications; the following pseudo-grammar provides a sketch:

$$\begin{array}{ll} expr & ::= Var \mid app \mid (expr). \\ app & ::= \tau(x_1, \ldots, x_n) \mid Func(expr_1, \ldots, expr_k). \\ Var & ::= \{u, v, w, x, y \ldots\}. \\ Func & ::= \{f, g, h, \wedge, \vee, \neg \ldots\}. \end{array} \quad (6)$$

(Note that τ applications are normalized so τ is only applied to variables.) Let $\varphi_\tau[V]$ be a quantifier-free formula containing one or more applications of τ; V is the set of variables appearing in φ. We refer to a (first-order) *universally quantified property of* τ as a statement of the form:

Definition 1. ***Universally quantified property of*** τ

$$\forall x_1 \in Q_1 \ldots \forall x_k \in Q_k \quad \varphi_\tau[x_1, \ldots, x_k]. \quad (7)$$

where every variable x_i is universally quantified over an input syntax $Q_i = S(X_j)$. We write $\tau \vdash p$ if property p can be deduced from τ. For the remainder of this paper we deal with this restricted class of properties, which encompasses a number of important examples:

Example 3. **Static Correctness.** As in traditional programming languages, an instance of DSL syntax $x \in S(X)$ is not guaranteed to be semantically meaningful. A compiler performs static analysis, e.g. type-checking, to check that x is meaningful. Let $check_X(\cdot)$ be a predicate evaluating to true when a model is statically correct. Then a transformation $\tau : S(X) \to S(Y)$ preserving static correctness has the following property [1]:

$$\forall x \in S(X),\ check_X(x) \Rightarrow check_Y(\tau(x)). \tag{8}$$

This property generalizes to transformations with multiple inputs/outputs. Let π_i be a projection operator; when applied to an n-tuple it returns the i^{th} coordinate. Let τ be an transformation with n inputs and m outputs.

$$\begin{array}{l} \forall x_1 \in S(X_1) \\ \quad\vdots \\ \forall x_n \in S(X_n) \end{array} \quad \bigwedge_{1 \le i \le n} check_{X_i}(x_i) \Rightarrow \bigwedge_{1 \le j \le m} check_{Y_j}(\pi_j(\tau(x_1, \ldots, x_n))). \tag{9}$$

Example 4. **Behavioral Correspondence.** Let $\sim \subseteq S(X) \times S(Y)$ be a simulation relation over models of X and Y. A transformation preserves *behavioral correspondence* [14] if the output simulates the input, whenever the input is meaningful.

$$\forall x \in S(X),\ check_X(x) \Rightarrow check_Y(\tau(x)) \wedge x \sim \tau(x). \tag{10}$$

Behavioral correspondence can also be generalized to multiple inputs/outputs according to a family of simulation relations.

In order to develop general theorems about property preservation, some assumptions on abstract syntaxes are required. We shall make the assumption that every $S(X)$ is disjoint from every other $S(Y)$. Under this assumption, a property p must satisfy simple *compatibility* conditions before it can be lifted to another context. Properties that are incompatible with a context do not hold there. Of course, when deeper knowledge about of syntax structure is available, then these compatibility conditions can be augmented appropriately.

Definition 2. ***Compatible Properties.*** *Let $\tau \vdash p$ where τ has context $(\overline{X}, \overline{Y})$. Let τ' be a reused transformation with context $(\overline{X'}, \overline{Y'})$. A property p is compatible with the context $(\overline{X'}, \overline{Y'})$ if whenever a variable x appears as the i^{th} and j^{th} argument to a τ application then $X'_i = X'_j$.*

Example 5. Let $\tau : S(X) \times S(X) \to S(Y)$ and $\tau' : S(U) \times S(W) \to S(Z)$. Then the property $\forall x_1, x_2 \in S(X)\ \tau(x_1, x_2) = \tau(x_2, x_1)$ is not compatible in the new context because $S(U) \cap S(W) = \emptyset$.

3.3 A General Scheme for Property Preserving Reuse

We wish to determine if all compatible properties satisfied by τ are also satisfied by τ'. This is accomplished by establishing a behavioral equivalence between τ' and τ, which we call *lifting equivalence*. Assume $\tau : S(X) \to S(Y)$. We say τ' *lifts* τ if the following procedures are equivalent:

1. Calculate $y' = \tau'(x')$, and then rewrite $y' \in S(Y')$ to $y \in S(Y)$.
2. Rewrite $x' \in S(X')$ to $x \in S(X)$, and then calculate $y = \tau(x)$.

If τ' lifts τ, then τ' can be viewed as syntactic rewriting step followed by an application of τ. This scheme requires fixing a class $\mathcal{T}_{rw}$ of transformations, which we call *rewriting procedures.* Let $\mathcal{T}_{rw}(\overline{X'}, \overline{X})$ be the (possibly empty) subset of rewriting procedures from $\overline{X'}$ to $\overline{X}$. Formally, τ' lifts τ if for every rewriting procedure Λ on the inputs, there exists a rewriting procedure Γ on the outputs such that the diagram in Figure 1 commutes. In other words, there is no wrong choice for Λ. To simplify construction of rewriting procedures, $\mathcal{T}_{rw}$ must satisfy a decomposition criterion:

Definition 3. ***Class of Rewriting Procedures.*** *A class of rewriting procedures $\mathcal{T}_{rw}$ is a class of functions of the form $\Lambda : S(X'_1) \times \ldots \times S(X'_n) \to S(X_1) \times \ldots \times S(X_n)$. Every Λ can be decomposed into a direct product of unary rewrites:*

$$\Lambda = \langle \Lambda_1, \ldots, \Lambda_n \rangle \text{ and } \Lambda_i : S(X'_i) \to S(X_i) \in \mathcal{T}_{rw}. \tag{11}$$

In other words, $\Lambda(x'_1, \ldots, x'_n)$ can be calculated by point-wise rewriting each x'_i with Λ_i. The decomposition also agrees on how to perform rewrites: If components Λ_i and Λ_j have the same signature, then they are the same unary rewriting procedure.

Definition 4. ***Lifting Equivalence.*** *Let $\mathcal{T}_{rw}$ be a class of rewriting procedures. Then τ' lifts τ if both transformations have n-inputs/m-outputs and:*

$$\forall \Lambda \in \mathcal{T}_{rw}(\overline{X'}, \overline{X})\ \exists \Gamma \in \mathcal{T}_{rw}(\overline{Y'}, \overline{Y}) \quad \tau \circ \Lambda = \Gamma \circ \tau'. \tag{12}$$

Claim. If $\tau \vdash p$, τ' lifts τ, and p is compatible with τ', then $\tau' \vdash p'$ where p' is constructed by the following procedure:

1. Pick any Γ and Λ satisfying Equation (12).
2. Replace every occurrence of $\tau(x_{c_1}, \ldots, x_{c_n})$ in p with $\Gamma(\tau'(x'_{c_1}, \ldots, x'_{c_n}))$.
3. Replace every remaining occurrence of x_i with $\Lambda_{x_i}(x'_i)$ where Λ_{x_i} is any well-typed unary rewrite from the decomposition of Λ.
4. Quantify each variable x'_i over a well-typed $Q'_i = S(X'_j)$, which must exist.

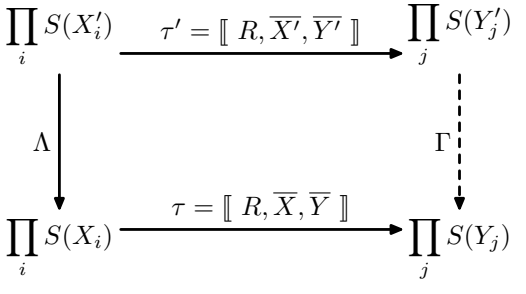

Fig. 1. A commuting diagram for lifting equivalence

We denote this replacement procedure by:

$$\forall x'_1 \in Q'_1, \ldots, \forall x'_k \in Q'_k \quad \varphi[x_1/\Lambda_{x_1}(x'_1), \ldots, x_k/\Lambda_{x_k}(x'_k), \tau/(\Gamma \circ \tau')]. \tag{13}$$

where $\varphi[x_1, \ldots, x_k]$ is the original formula appearing in p.

Example 6. **Lifting Static Correctness.** Given $\tau : S(X) \to S(Y), \tau' : S(X') \to S(Y')$. If τ' lifts τ and τ preserves static correctness, then p' becomes:

$$\forall x' \in S(X'),\ check_X(\Lambda(x')) \Rightarrow check_Y(\Gamma(\tau'(x'))). \tag{14}$$

Theorem 1. ***Property lifting.*** *If $\tau \vdash p$, τ' lifts τ, and p is compatible with τ, then $\tau' \vdash p'$ where p' is constructed according to (13). We say p' is a* lifting *of p.*

Proof. Observe that for every variable x_i quantified over Q_i there is at least one component $\Lambda_{x_i} : S(X') \to Q_i$. This is due to the compatibility condition (Definition 2) and the requirement that variables are quantified over input syntaxes (Definition 1). Since p holds for all values of x_i, replace every occurrence of x_i with $\Lambda_{x_i}(x'_i)$ where x'_i is a fresh variable. Each x'_i is quantified over $\mathbf{dom}\ \Lambda_{x_i}$, which we denote Q'_i yielding the property:

$$\forall x'_1 \in Q'_i, \ldots, \forall x'_k \in Q'_k \quad \varphi_\tau[x_1/\Lambda_{x_1}(x'_1), \ldots, x_k/\Lambda_{x_k}(x'_k)]. \tag{15}$$

This property still has occurrences of τ. However, since the τ applications in p were normalized to $\tau(x_{c_1}, \ldots, x_{c_n})$, then every application in (15) has the form $\tau(\Lambda_{x_{c_1}}(x'_{c_1}), \ldots, \Lambda_{x_{c_n}}(x'_{c_n}))$. This can be rewritten $(\tau \circ \Lambda)(x'_{c_1}, \ldots, x'_{c_n})$. Applying Equation (12), this is equivalent to $(\Gamma \circ \tau')(x'_{c_1}, \ldots, x'_{c_n})$, which yields $\Gamma(\tau'(x'_{c_1}, \ldots, x'_{c_n}))$. Thus, we obtain a property over τ' according to (13). □

3.4 Summary of the Approach

Our approach relies on a class of rewriting procedures as a basis for comparing an original transformation with its reused version. Given τ' and τ, our algorithms characterize the set of rewriting procedures for reconciling the context of τ' with the context of τ. If no procedures can be found, then the contexts are too different and no guarantees can be provided about property preservation. If rewriting procedures exist, then it must be ensured that diagram 1 commutes for any choice of procedure, guaranteeing that lifted properties hold regardless of this choice. If this can be verified, then every compatible property p holding for τ also holds for τ' (in the sense of Theorem 1) even if p is not explicitly known to hold for τ. This is due to the behavioral equivalence that exists between the two transformations.

The effectiveness of this approach depends crucially on the choice for $\mathcal{T}_{rw}$. If the class is too complicated, then it may be computationally prohibitive to verify that τ' lifts τ. If the class is too simple, then either most contexts cannot be reconciled or occurrences of rewriting procedures cannot be eliminated from lifted properties. In other words, lifted properties may indirectly depend on the

original context through the rewriting procedures. Fortunately, for some lifted properties it is possible to remove these occurrences, thereby obtaining an equivalent property with no dependency on the original context. For the remainder of this paper we show a reasonable choice for $\mathcal{T}_{rw}$ that leads to computationally efficient algorithms and to lifted properties where elimination of rewriting procedures can be automated.

4 Implementing Lifting Analysis

For the remainder of this paper we apply these techniques to strongly-typed rule-based systems where models are instances of recursive data types. We develop a useful class of rewriting procedures for algebraic data types, called *collapsing morphisms*.

4.1 Example: Reuse in FORMULA

We motivate the following sections by illustrating lifting analysis in our FORMULA framework, beginning with the classic *non-deterministic finite state automata* (NFA) abstraction specified with FORMULA. The left side of Figure 2 shows the syntax and static semantics for the NFA DSL. The domain keyword declares a DSL called NFA (line 1). DSL syntax is defined via a set of record constructors. For example, line 3 defines a record constructor State, which takes an integer ID and returns a State record with that ID. Record constructors can have more complex type constraints; e.g. the Transition constructor takes two State records an an Event record as input. Equality is defined over records; two records are the same if both were constructed by the same constructor using

```
1.  domain NFA
2.  {
3.    State : (id: Integer).
4.    Event : (id: Integer).
5.    [relation]
6.    Transition : (src: State,
7.     trg : Event, dst: State).
8.    [relation]
9.    Initial    : (state: State).
10.
11.   //At least one initial state.
12.   conforms :? i is Initial.
13. }.
14.
15.
```

```
16. domain ProdNFA
17. {
18.   State    : (id: StateLbl).
19.   StateLbl : Integer + Pair.
20.   Pair     : (p1: ProdLbl,
21.               p2: ProdLbl),
22.   ProdLbl  : State + Pair.
23.   Event    : (id: Integer).
24.   [relation]
25.   Transition : (src: State,
26.    trg : Event, dst: State).
27.   [relation]
28.   Initial    : (state: State).
29.   conforms   :? i is Initial.
30. }.
```

Fig. 2. (Left) Simple automata DSL, (Right) Product automata DSL

the same arguments (i.e. structural equality). An instance x of DSL syntax is a finite set of finite records:

$$x = \{State(1), State(2), Event(3), Transition(State(1), Event(3), State(2))\} \tag{16}$$

A FORMULA specification also contains static semantics for the DSL. The annotations on lines 5, 8 require Transition and Initial records to behave like relations over states and events. Line 12 is an explicit conformance rule requiring at least one initial state (i.e. at least one Initial record).

The NFA DSL contains just enough elements to express the most basic of NFAs. For example, it is inconvenient to express products of NFAs, because the State constructor cannot hold the IDs of product states. The ProdNFA domain remedies this situation by defining states with more complex IDs (line 18). Now, IDs are instances of the StateLbl type, which is a union of the Integer and Pair types. In turn, a Pair constructor accepts either a State or another Pair. Consequently, State is a recursive data type permitting IDs such as:

$$State\Big(Pair\big(Pair(State(1), State(2)), Pair(State(3), State(4))\big)\Big) \tag{17}$$

At this point, the FORMULA compiler does not know that these two DSLs are related.

The two DSLs can be explicitly related by a transformation taking two NFAs and returning their synchronous product. Such a transformation has the signature:

```
transform SProd (NFA as in1, NFA as in2) returns (ProdNFA as out) {...}
```

The identifiers in1, in2, and out are special variables that hold the input and output models during execution of the transformation. Two NFAs can be composed with SProd, but further composition is not possible since SProd does not accept ProdNFA models as inputs. Intuitively, the rules defining SProd should behave similarly in the context $\overline{X'} = [\mathsf{ProdNFA}, \mathsf{ProdNFA}]$, $\overline{Y'} = [\mathsf{ProdNFA}]$. This intuition can be stated using the following one line declaration.

```
transform SProd2 lifts SProd overrides (ProdNFA as in1, ProdNFA as in2).
```

The SProd2 transformation interprets the rules from SProd in a new context according to the list of overrides. FORMULA accepts this declaration if it can be verified that SProd2 lifts SProd, in which case the lifted transformation also lifts properties. Otherwise, an error is emitted.

The lifting analysis employs a class of rewriting procedures that we call *collapsing morphisms*. This class allows automatic elimination of rewrites appearing in lifted properties. For example, we know that:

$$\forall x_1, x_2 \in S(NFA),\ \#states(SProd(x_1, x_2)) = \#states(x_1) \times \#states(x_2)\,.$$

Assume SProd2 lifts SProd, then the lifted property is:

$$\forall x'_1, x'_2 \in S(ProdNFA),\ \#states(\Gamma(SProd2(x'_1, x'_2))) = \begin{array}{l}\#states(\Lambda_{x_1}(x'_1)))\times \\ \#states(\Lambda_{x_2}(x'_2)))\end{array}.$$

If Γ and Λ are collapsing morphisms, then the rewrites can be immediately eliminated, yielding:

$$\forall x'_1, x'_2 \in S(ProdNFA),\ \#states(SProd2(x'_1, x'_2)) = \#states(x'_1) \times \#states(x'_2)\,.$$

4.2 Collapsing Morphisms as Rewriting Procedures

Instances of algebraic data types with structural equality can be formalized as either *terms* over a *term algebra* or as *ordered trees* [24]. We describe the ordered tree representation as it simplifies description of algorithms. An ordered tree is a tree where the children of node v are ordered 1 to k_v. A record s instantiated by $f(s_1, \ldots, s_n)$ produces an ordered tree where the root is labeled by the constructor f and the i^{th} child is the root of the i^{th} subtree s_i. Syntactically, a model $x \in S(X)$ is a set of ordered trees; the left-hand side of Figure 3 shows the set of ordered trees corresponding to (16) from the previous section. (By convention children are drawn in order from left to right.) Note that every internal node must be labeled by a record constructor and every leaf node must be a value, such as the integer 3. (We treat nullary constructors as user-defined values.)

From this perspective, rewriting procedures must reconcile the legal trees of $S(X')$ with the legal trees of another syntax $S(X)$. Our approach is to preserve subtrees that are common to both syntaxes, while collapsing new types of subtrees from $S(X')$ into arbitrary values. The right side of Figure 3 illustrates this. On one hand, there is a complex State record from the ProdNFA domain. The rewriting procedure Λ transforms the root node into an equivalent node in the NFA domain, because the State constructor is common to both domains. However, the ID of the State record is rooted by a Pair node, which does not exist in the NFA domain. This entire subtree is collapsed into a single value $\sigma \in \Sigma_X$, where Σ_X is the set of all values that can appear in the trees of X. The result is a well-typed tree in the original syntax that preserves as much common structure as possible, and disguises foreign subtrees as values. A tree s is *legal* in $S(X)$ if $s \in \Sigma_X$ or s is rooted by the constructor f and its children satisfy the type-constraints of this constructor. Let $Trees(X)$ be the set of all legal finites trees of DSL X, then $S(X) = \mathcal{P}(Trees(X))$ is all finite sets of such trees.

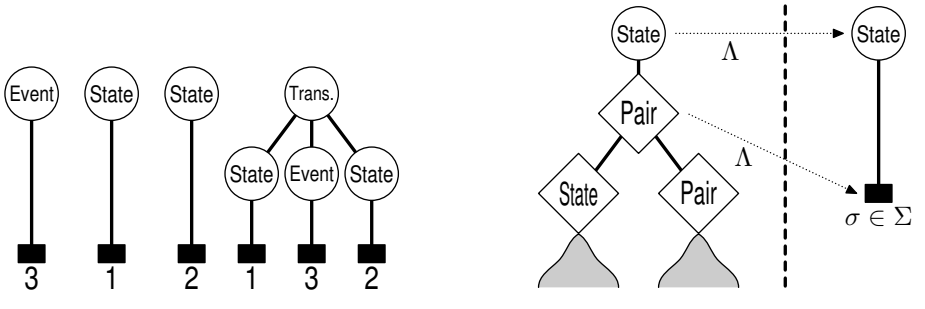

Fig. 3. (Left) Instance of syntax as set of ordered trees, (Right) Action of collapsing morphism Λ on trees

The motivation for this class of rewriting procedures is based on the following observation: Many transformation rules do not require full examination of record structure. Consider the rule for constructing product states; in pseudo-code:

Pattern: **Match** State s_1 from $input_1$, **match** State s_2 from $input_2$.
Action: For every match, **create** State(x) where x = Pair(s_1.id, s_2.id); add it to the output.

This rule uses the IDs to distinguish one state from another and to name product states, but the internal structural of the IDs is not important. Thus, we expect this rule to behave equivalently whether IDs are integers or trees of pairs. In fact, given two ProdNFA models as inputs, the product states could be calculated by first collapsing state IDs into distinct integers and then running the original SProd transformation. At the end replace these integers with their corresponding subtrees to obtain to correct result.

Rewriting procedures of this form do not exist between all pairs $(S(X'), S(X))$. It must be the case that: (1) Record constructors common to both syntaxes have the same arity, though type-constraints can differ. (2) Every legal finite tree in $S(X')$ can be rewritten into a legal finite tree in $S(X)$, taking into account collapsing of foreign subtrees. (3) For every *finite set* of legal finite trees the collapsing action must be in one-to-one correspondence. We formalize this in two parts by first characterizing morphisms over trees, and then generalizing these to finite sets of trees. Let $Cons(X)$ be the set of record constructors for DSL X.

Definition 5. ***Collapsing Tree Morphism.*** *Given X' and X such that common constructors agree on arity, then a collapsing tree morphism $\lambda : Trees(X') \to Trees(X)$ has the following properties:*

1. *Common values are fixed: $\lambda(s) = s$ if $s = \sigma \in (\Sigma_X \cap \Sigma_{X'})$.*
2. *A tree rooted with a shared constructor is preserved: $\lambda(s) = f(\lambda(s_1), \ldots, \lambda(s_n))$ if $s = f(s_1, \ldots, s_n)$ and $f \in (Cons(X) \cap Cons(X'))$.*
3. *All other trees are collapsed to a value: $\lambda(s) = \sigma$ and $\sigma \in \Sigma_X$ if neither (1) nor (2) apply.*

If collapsing tree morphisms exist, then trees from X' can always be rewritten to X. However, this does not guarantee that distinct trees can always be collapsed into distinct constants, which requires a finite inverse condition.

Definition 6. ***Collapsing Morphism.*** *Given X' and X, then a collapsing morphism $\Lambda : S(X') \to S(X)$ maps finite sets of trees so that distinct collapsed subtrees are mapped to distinct values. Specifically, applying Λ to a set x' is equivalent to extending a collapsing tree morphism over this set:*

$$\forall x' \in S(X')\ \exists \lambda_{x'}, \quad \Lambda(x') = \bigcup_{s \in x'} \lambda_{x'}(s). \tag{18}$$

Every $\lambda_{x'}$ is one-to-one for the subtrees of x':

$$\forall s_1, s_2 \in x'\ \forall t_1 \sqsubseteq s_1, t_2 \sqsubseteq s_2 \quad (\lambda_{x'}(t_1) = \lambda_{x'}(t_2)) \Rightarrow (t_1 = t_2), \tag{19}$$

where $t \sqsubseteq s$ indicates that t is an ordered subtree of s.

Collapsing morphisms establish global relationships between syntaxes, so it is not surprising that they can often be eliminated from lifted properties. In the interest of space, we present one example of this elimination. Given a DSL X, a counting function $\#f(x)$ counts the number of trees rooted by constructor $f \in Cons(X)$ occurring in the set x. (We already made use of the $\#State$ counting function.)

Theorem 2. ***Eliminating Rewrites from Counting Functions.*** *Given a subformula $\#f(\Lambda(x'))$, where $x' \in S(X')$ and Λ is a collapsing morphism from $S(X')$ to $S(X)$:*

$$\#f(\Lambda(x')) = \begin{cases} 0 \text{ if } f \notin Cons(X') \\ \#f(x') \text{ otherwise} \end{cases} \tag{20}$$

As a final note, we have described unary collapsing morphisms. An arbitrary rewrite Λ is decomposable into a direct product of unary rewrites, so these results generalize immediately.

4.3 Calculating Collapsing Morphisms

We now turn our attention to calculating the set of collapsing morphisms, $CM(X', X)$, between DSLs X' and X. Our algorithm represents $CM(X', X)$ by a mapping from types declared in X' to type declarations compatible with X such that every collapsing tree morphism λ must respect this type map. If the algorithm fails to map every type in X', then no λ exists and $CM(X', X) = \emptyset$. The success of this algorithm guarantees the existence of λ's, but it does not guarantee the existence of a $\Lambda : S(X') \to S(X)$ satisfying the finite inverse condition. Fortunately, it can be constructively shown that Λ's exist by solving a *maximum bipartite matching problem* between a finite number of trees in $Trees(X')$ and $Trees(X)$.

A type declaration d can be any of the following: (1) A record constructor f: $(T_1,\ldots,T_n)$. (2) A finite enumeration of values e: $\{\sigma_1,\ldots,\sigma_n\}$. (3) A (non-disjoint) union of types u: $T_1 + \ldots + T_n$. Each T_i is the name of some type, and all DSLs share (order-sorted) infinite alphabets of values, e.g. T_{integer}, T_{string}. The type T_{basic} is the set of all values. Every type accepts a set of ordered trees, denoted $Trees(T)$. In our algorithms we use the fact that inclusion and equality testing between types is decidable and the type system is closed under union, intersection, and complement. In fact, every type is equivalent to a *tree automaton* that accepts exactly the set $Trees(T)$, so operations on types correspond to operations on tree automata [24]. (The details of tree automata algorithms are outside the scope of this paper.) Algorithm 1 tries to build a re-declaration map, called *redecl*, from type names in X' to declarations/alphabets compatible with X.

If Algorithm 1 succeeds, then *redecl* characterizes how every collapsing tree morphism behaves. Algorithm 2 ensures that the finite inverse condition can hold by checking if an invertible λ exists even under worst case conditions. The algorithm constructs a matching problem whenever a finite non-enumeration type T' must collapse into another finite type T. It succeeds if this matching

Algorithm 1. Compute Re-declaration Map

for all $f' \in (Cons(X') - Cons(X))$ **do**
  update $redecl(T_{f'}) := T_{\mathsf{basic}}$
for all $T' \in Types(X')$ where $T' \subseteq T_{\mathsf{basic}}$ **do**
  if T' is declared to be a finite enumeration $e' : \{\sigma_1, \ldots, \sigma_n\}$ **then**
    update $redecl(T') := e : \{\sigma_1, \ldots, \sigma_n\}$
  else
    update $redecl(T') := T_a$ // T' must be a built-in alphabet T_a
for all $f \in (Cons(X') \cap Cons(X))$ **do**
  lookup declarations $d' = f : (T'_1, \ldots, T'_n)$ and $d = f : (T_1, \ldots, T_n)$
  for all pairs (T'_i, T_i) **do**
    for all $T'' \in Types(X')$ where $T'' \subseteq T'_i$ and $T'' \nsubseteq T_i$ **do**
      if $T'' \subseteq T_{\mathsf{basic}}$ or $T'' = T_g$ where $g \in (Cons(X') \cap Cons(X))$ **then**
        return false
      else if $T'' = T_{g'}$ where $g' \in (Cons(X') - Cons(X))$ **then**
        let $T_{old} = redecl(T_{g'})$
        update $redecl(T_{g'}) := (T_{old} \cap T_i)$
return true

problem has a perfect matching, which is decidable in polynomial time (e.g. via the *Hopcroft-Karp* algorithm [25]). Note that $Values(X')$ is the set of all values that could appear as an argument to any constructor of X'. A full complexity analysis is outside the scope of this paper. However, the following theorem is immediate from the algorithm:

Algorithm 2. Check Finite Inverses

update $match := \{\}$ // Initialize a map called *match* to the empty map.
for all (T', T) where $redecl(T') = T$ **do**
  if $|T'| > |T|$ **then**
    return false
  if T' is a finite non-enumeration type and T is a finite enumeration **then**
    for $s \in Trees(T')$ **do**
      if s is not in the domain of *match* **then**
        update $match(s) := (Trees(T) - Values(X'))$
      else
        let $match_{old} = match(s)$
        update $match(s) := (Trees(T) \cap match_{old})$
return HasPerfectMatching(*match*)

Theorem 3. ***Construction of Collapsing Morphisms.*** *The set of collapsing tree morphisms $CM(X', X)$ can be characterized with a polynomial number of type comparisons (e.g. tree automata operations) and a maximum bipartite matching problem of size $c(|Types(X')| + |Types(X)|)$ where c is the size of the largest finite enumeration.*

Surprisingly, the calculation of collapsing morphisms is the primary task to check if τ' lifts τ. After $CM(X', X)$ is calculated, static analysis determines if the diagram in Figure 1 commutes. This verification can be accomplished fairly easily, because the compiler knows that τ' and τ were generated by the same rule set. Static analysis examines the interpretation of each rule in the new and original contexts, and checks if any rule patterns are sensitive to the choice of Λ. All FORMULA rules are strongly typed during compile time, so a simple type comparison is required to test if a pattern might be sensitive to this choice.

5 Conclusion

We presented a novel framework for deciding if a reused transformation preserves properties. The key idea is to relate a reused transformation with its original version through an automatically deducible rewriting procedure. A reused transformation preserves compatible properties if it is behaviorally equivalent to a rewrite followed by the original transformation. We formalized a class of useful rewriting procedures, called collapsing morphisms, which can be automatically derived. Furthermore, properties lifted using collapsing morphisms are amenable to automatic elimination of rewrites. These procedures have been implemented in our FORMULA framework.

References

1. Jackson, E.K., Sztipanovits, J.: Formalizing the structural semantics of domain-specific modeling languages. Software and Systems Modeling (2008)
2. Jackson, E.K., Seifert, D., Dahlweid, M., Santen, T., Bjørner, N., Schulte, W.: Specifying and composing non-functional requirements in model-based development. In: Bergel, A., Fabry, J. (eds.) Software Composition. LNCS, vol. 5634, pp. 72–89. Springer, Heidelberg (2009)
3. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
4. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3), 214–234 (2007)
5. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. 152, 125–142 (2006)
6. Taentzer, G.: AGG: A tool environment for algebraic graph transformation. In: Münch, M., Nagl, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 481–488. Springer, Heidelberg (2000)
7. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
8. de Lara, J., Vangheluwe, H.: Defining visual notations and their manipulation through meta-modelling and graph transformation. J. Vis. Lang. Comput. 15(3-4), 309–330 (2004)
9. Ehrig, H., Ehrig, K., Ermel, C.: Evolution of model transformations by model refactoring. In: Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT 2009 (2009)

10. Levendovszky, T., Balasubramanian, D., Narayanan, A., Karsai, G.: A novel approach to semi-automated evolution of dsml model transformation. In: 2nd International Conference on Software Language Engineering, SLE (2009)
11. Meertens, L.: Paramorphisms. Formal Aspects of Computing 4, 413–424 (1992)
12. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: FPCA, pp. 124–144 (1991)
13. Achatz, K., Schulte, W.: Massive parallelization of divide-and-conquer algorithms over powerlists. Sci. Comput. Program. 26(1-3), 59–78 (1996)
14. Narayanan, A., Karsai, G.: Towards verifying model transformations. Electr. Notes Theor. Comput. Sci. 211, 191–200 (2008)
15. Varró, D.: Automated formal verification of visual modeling languages by model checking. Journal of Software and Systems Modeling 3(2), 85–113 (2004)
16. Ehrig, H., Ermel, C.: Semantical correctness and completeness of model transformations using graph and rule transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 194–210. Springer, Heidelberg (2008)
17. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 347–361. Springer, Heidelberg (2008)
18. Bisztray, D., Heckel, R., Ehrig, H.: Compositionality of model transformations. Electr. Notes Theor. Comput. Sci. 236, 5–19 (2009)
19. Balasubramanian, D., Narayanan, A., van Buskirk, C.P., Karsai, G.: The Graph Rewriting and Transformation Language: GReAT. ECEASST 1 (2006)
20. de Lara, J., Guerra, E.: Pattern-based model-to-model transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 426–441. Springer, Heidelberg (2008)
21. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with maude. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 54–73. Springer, Heidelberg (2009)
22. Cortellessa, V., Gregorio, S.D., Marco, A.D.: Using atl for transformations in software performance engineering: a step ahead of java-based transformations? In: WOSP, pp. 127–132 (2008)
23. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Monticore: a framework for the development of textual domain specific languages. In: ICSE Companion, pp. 925–926 (2008)
24. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications, `http://www.grappa.univ-lille3.fr/tata` (2007) (release October 12, 2007)
25. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. 2(4), 225–231 (1973)