Specification and Verification of Graph-Based Model Transformation Properties*

Gehan M. K. Selim[1], Levi Lúcio[2], James R. Cordy[1], Juergen Dingel[1], and Bentley J. Oakes[2]

[1] School of Computing, Queen's University, Kingston ON K7L2N8, Canada,
{gehan, cordy, dingel}@cs.queensu.ca
[2] School of Computer Science, McGill University, Montreal QC H3A2A7, Canada,
levi@cs.mcgill.ca, bentley.oakes@mail.mcgill.ca

Abstract. We extend a previously proposed symbolic model transformation property prover for the DSLTrans transformation language. The original property prover generated the set of *path conditions* (i.e., symbolic transformation executions), and verified atomic contracts (constraints on input-output model relations) on these path conditions. The prover evaluated atomic contracts to yield either *true* or *false* for the transformation when run on any input model. In this paper we extend the prover such that it can verify atomic contracts and more complex properties composed of atomic contracts. Besides demonstrating our prover on a simple transformation, we use it to verify different kinds of properties of an industrial transformation. Experiments run on this transformation using our prover provide results that are two orders of magnitude better than another verification tool we have evaluated in previous research.

Keywords: MDD, model transformation, verification, property prover.

1 Introduction

In Model-Driven Development (MDD), *models* are the basic blocks of software development, and *model transformations* are used to map between models conforming to different metamodels. Given their key role in MDD, verification of transformations is becoming of increasing interest to researchers [2, 16].

This study investigates verifying properties of transformations implemented in the graph-based model transformation language DSLTrans [7]. DSLTrans is non-Turing complete, i.e., DSLTrans cannot specify transformations that require unbounded loops (e.g., simulation transformations). We extend a symbolic model transformation property prover for DSLTrans [14, 12] that was previously limited to verifying atomic contracts (i.e., constraints on input-output model relations). The extension we present in this paper supports a more expressive property language that facilitates verifying atomic contracts and compositions of atomic contracts in the form of propositional logic formulae. Moreover, our prover now handles rules that overlap in their application.

* This work is supported in part by NSERC, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp.

The contribution of this study, at a high level, is extending a DSLTrans property prover that is *input-independent* [2], i.e., verification results generated by the prover hold for all possible inputs. Our specific contributions are:

- We describe how our prover currently handles overlapping rules (Section 4).
- We introduce our new property language, and show how it can be used to express commonly occurring properties, e.g., multiplicity invariants. (Section 5).
- We apply our extended prover to an industrial case study [18] (Section 6).
- We demonstrate how our extensions of the prover led to a two orders of magnitude improvement in execution time over the verification tool we used in another study [17]. We also discuss the strengths and limitations of our prover (Section 7).

This study adds to the state of the art (as discussed in Section 8) and is useful to transformation verification research in general. We provide some evidence for our prover's scalability and usefulness since verification using our prover need not be redone for every input. Thus, we motivate researchers to adopt our prover. Moreover, users of languages other than DSLTrans can benefit from our study in two ways: (1) the study can be used as a guide to develop an input-independent verification technique for any language; (2) plug-ins can be developed to convert transformations in other languages to DSLTrans to be able to use our prover.

The rest of this paper is organized as follows: Section 2 summarizes DSLTrans and it's simplest properties; Section 3 overviews our prover's architecture; Section 4 describes path condition generation; Section 5 discusses our prover's verification technique; Section 6 demonstrates an industrial case study; Section 7 discusses our prover's strengths and limitations; Section 8 reviews related work; and Section 9 concludes and presents future work.

2 The DSLTrans Model Transformation Language

DSLTrans [7] is a graph-based transformation language that can be used to specify out-place, model transformations that are confluent and terminating by construction. Transformation rules in DSLTrans are constructive – elements can be created but not deleted. The semantics of DSLTrans (currently defined using set theory) are in-line with, and can be defined using, classical pushout approaches. We demonstrate DSLTrans using a simple transformation as a running example.

Figs. 1 and 2 present two metamodels used to describe different views of a set of persons. The 'Household Language' represents persons as members of families which in turn form a set of households. The 'Community Language' represents persons as men or women who belong to a community.

Fig. 3 presents a DSLTrans transformation that aims to transform family members in the 'Household Language' (source metamodel) into men and women of a community in the 'Community Language' (target metamodel). In what follows, we refer to the transformation in Fig. 3 as the *Persons* transformation.

A DSLTrans transformation is composed of an ordered set of layers (e.g., 'TopLevel', 'FamilyMembersToGender', and 'BuildCommunityOfPersons' layers in Fig. 3) that are executed sequentially. A layer consists of a set of transformation rules that execute in a non-deterministic order but produce a deterministic result. Each rule is a pair (*MatchModel*, *ApplyModel*) where *MatchModel* is a pattern of source metamodel elements and *ApplyModel* is a pattern of target

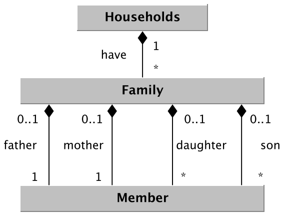

Fig. 1. Household Language

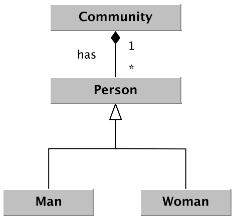

Fig. 2. Community Language

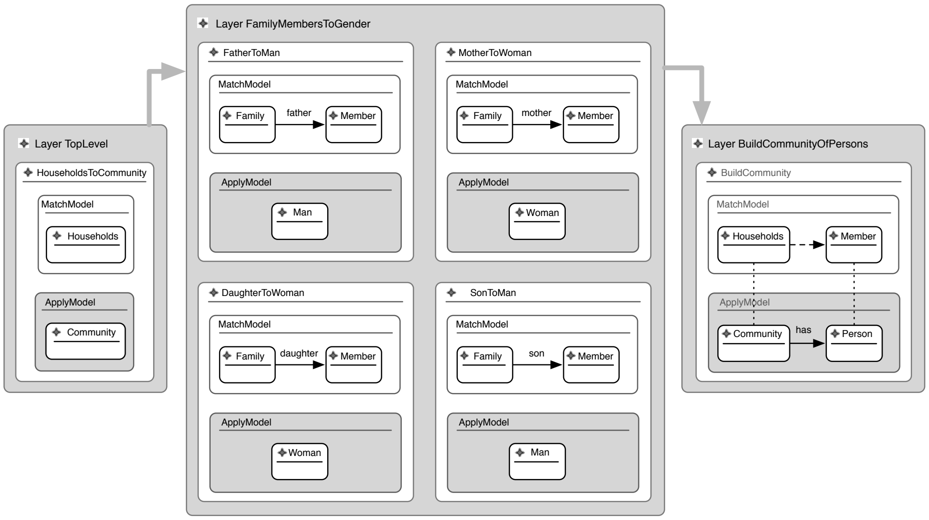

Fig. 3. The *Persons* Transformation expressed in DSLTrans.

metamodel elements. For example, the MatchModel of the 'HouseholdsToCommunity' rule in the 'TopLevel' layer (Fig. 3) has one 'Households' class from the 'Household Language' and the ApplyModel has one 'Community' class from the 'Community Language'. This means that 'Households' input model elements will be transformed into 'Community' output model elements.

When a DSLTrans rule executes, *traceability links* are created between each element in the rule's MatchModel and each element in the ApplyModel. These are used to keep track of which output elements came from which input elements.

We describe some DSLTrans constructs that are used to build the MatchModel of a DSLTrans rule. More DSLTrans constructs can be found in [7, 12].

- *Match Elements* are variables typed by source metamodel classes that can assume as values instances of that class from the input model. An example of a match element is the 'Family' element in the 'FatherToMan' rule (Fig. 3). Match elements can be of two types: *Any* match elements are bound to all matching instances in the input model, and *Exists* match elements are bound to only one (deterministic) matching instance in the input. All match elements in Fig. 3 are of type *Any*.
- *Attribute Conditions* are conditions on the attributes of a match element.
- *Direct Match Links* are links between two match elements that are typed by labelled relations of the source metamodel. These links can assume as values links having the same label in the input model.

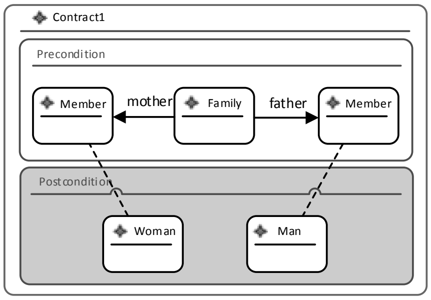

Fig. 4. Contract1; should hold.

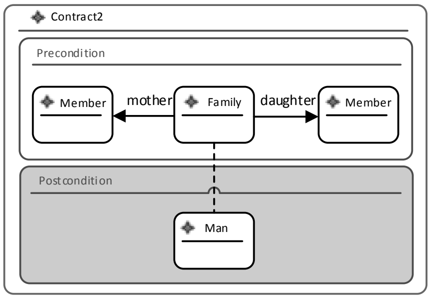

Fig. 5. Contract2; should not hold.

- *Indirect Match Links* represent a path of containment associations between the linked match elements. For example, an indirect match link appears in the 'BuildCommunity' rule as a horizontal, dashed arrow between match elements.
- *Backward Links* link elements of the MatchModel and the ApplyModel of a rule, e.g., backward links are used in the 'BuildCommunity' rule and are denoted as vertical, dashed lines. Backward links are used to refer to traceability links between input and output model elements that are generated by the rules of previous layers.

Similar constructs can be used to build a rule's ApplyModel, as shown in Fig. 3.

- *Apply elements* are variables typed by target metamodel classes and linked by *apply links*. Apply elements that are not connected by backward links create output elements of the same type each time the MatchModel is found in the input. Apply elements that are connected by backward links are handled differently, e.g., 'BuildCommunity' rule connects 'Community' and 'Person' output elements that were formerly created from 'Households' and 'Member' input elements with a 'has' link.
- Apply elements can have *apply attributes* that can be set from references to one or more attributes of match elements.

***AtomicContracts* in DSLTrans:** An *AtomicContract* is the simplest property that can be expressed in our prover. Each *AtomicContract* is a pair (*pre, post*) that specifies a property of the form: "if the input model satisfies the precondition *pre*, then the output model should satisfy the postcondition *post*". A precondition is a constraint on the transformation's input model in the form of a structural relation between input model elements. Similarly, a postcondition is a constraint on the transformation's output in the form of a structural relation between output elements. Preconditions and postconditions are expressed using the same constructs as rules. Postconditions may also have traceability links to link postcondition elements to precondition elements. This signifies that the property will only match an output element that was previously created from an input element. The formal definition of an *AtomicContract* can be found in [12]. Figs. 4 and 5 demonstrate two *AtomicContracts* for the *Persons* transformation. Fig. 4 is interpreted as follows: "a mother and a father in a family will always be transformed to a woman and a man". Fig. 5 is interpreted as follows: "a family including a mother and a daughter will always be transformed to a man". Our prover should verify that the *AtomicContract* in Fig. 4 will always hold for the *Persons* transformation, while the *AtomicContract* in Fig. 5 will not always hold (with a supporting counterexample).

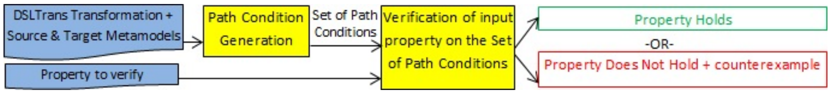
DSLTrans Transformation + Source & Target Metamodels
Property to verify
Path Condition Generation
Set of Path Conditions
Verification of input property on the Set of Path Conditions
Property Holds
-OR-
Property Does Not Hold + counterexample

Fig. 6. The architecture of our symbolic model transformation property prover.

3 The Symbolic Model Transformation Property Prover

Fig. 6 demonstrates our property prover's final architecture. Our prover takes four inputs: the DSLTrans transformation of interest, the transformation's source and target metamodels, and the property to verify. Verification is then carried out in two steps, as shown in Fig. 6. First, the prover generates the set of *path conditions* representing all possible executions of the input transformation (Section 4). Then, the prover verifies the input property on the generated set of path conditions and renders the property to be either *true* or *false* (with a counter example) for the transformation when run on any input model (Section 5).

4 Generating the Set of Path Conditions

Our property prover generates a set of *path conditions* that symbolically represent the possible transformation executions. For a transformation with n layers, our prover uses the transformation rules to build the path conditions in n iterations. In Fig. 7, we demonstrate how the path conditions for the *Persons* transformation are generated in iterations. We identify every rule in each layer of Fig. 3 with a pair of numbers, e.g., 4_2 corresponds to the fourth rule (ordered from top to bottom and then from left to right in Fig. 3) in the second layer (i.e., 'SonToMan' rule). We start off with the empty path condition, where we assume no transformation rule has been applied. To generate path conditions in iteration 1, the empty path condition is combined with all possible rule combinations of the first transformation layer. Similarly, to generate path conditions in iteration 2, each path condition from iteration 1 is combined with all *applicable* rule combinations of the second layer. A rule combination of the second layer that does not have backward links is always *applicable*, since it does not depend on rules from the first layer. Rule combinations of the second layer with backward links are combined with a path condition from iteration 1 only if the path condition generates the elements linked by backward links in the rule combination.

Each path condition thus accumulates a set of rules describing a possible path of rule applications through the transformation's layers. We refer to the accumulated MatchModels (or ApplyModels) of all the rules in a path condition as the path condition's *match pattern* (or *apply pattern*). Since our technique abstracts from how many times the rule executes for an input, a transformation rule only occurs once in each path condition. Thus, a path condition symbolically represents a set of concrete executions since each of the rules in a path condition can be concretely executed any number of times on an input model.

In Fig. 8, we show the path condition of the node with the dotted edge in Fig. 7. As shown from the numbers in the node, the path condition contains four combined rules (i.e., 'HouseholdsToCommunity', 'FatherToMan', 'MotherToWoman', 'BuildCommunity') and traceability links. When combining the rules, elements of the same type of the combined rules can be merged. This represents the fact that different rules may execute over the same input elements.

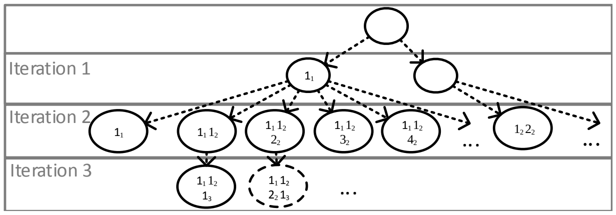

Fig. 7. Generation of the set of path conditions in iterations.

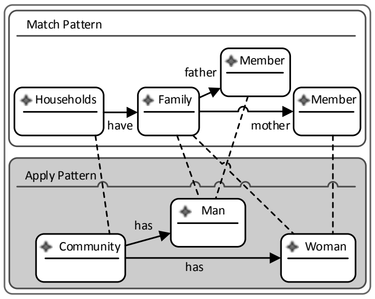

Fig. 8. A path condition of the Persons transformation.

Only the path conditions from the last iteration are returned as as the result since they capture all the possible *complete* transformation executions. Details on path condition generation can be found in [12].

Overlapping Rules: The industrial case study presented later in Section 6 uncovered an unforeseen transformation structure; *overlapping rules.* These can be defined as follows: whenever two rules in the same layer use match elements of the same metamodel classes of type *Any* or *Exists*, then the MatchModel of one rule syntactically *subsumes* the MatchModel of the other rule. For example, a rule having a MatchModel consisting of only an *Any* match element of class 'A' is subsumed by a MatchModel of another rule that consists of an *Exists* match element of class 'A' and an *Any* match element of class 'B'.

Our path condition generation algorithm was extended to handle overlapping rules. This extension led to a pronounced decrease in the number of generated path conditions in our case study, since a set of rules in a *subsumption* relation (described above) can often be merged into a smaller set of rules. Depending on whether rules overlap totally or partially, rule merge may be done before path condition generation or during path condition generation. For transformations with rule overlaps, this extension leads to an improved management of the combinatorial explosion in path condition generation [12].

5 Verification of the Property of Interest

We extended the technique proposed in [14] for verifying *AtomicContracts* of DSLTrans transformations to enable the verification of more complex properties. Our extended technique employs the following syntax and semantics.

Syntax: Our syntax is based on propositional logic. An *AtomicContract* (*pre,post*) is the smallest unit in our property language. A propositional formula can be built using one or more *AtomicContracts* and the operators $\neg_{tc}$ (*not*),

$\vee_{tc}$ (*or*), $\wedge_{tc}$ (*and*), and $\Longrightarrow_{tc}$ (*implication*), where *tc* stands for "transformation contract". Assuming that *(pre,post)* is an element of the set of *AtomicContracts AC*, the syntax of formulae is:

$$\varphi := \ (pre, post) \mid \neg_{tc}\varphi \mid \varphi \vee_{tc} \varphi \mid \varphi \wedge_{tc} \varphi \mid \varphi \Longrightarrow_{tc} \varphi \tag{1}$$

Free variables can occur in any element e of an *AtomicContract*'s pre/ post-condition. This occurrence binds the free variable to all the matches found for e within an instantiation of a MatchModel. Using the same free variable in different *AtomicContracts* allows these *AtomicContracts* to refer to the same matched element, e.g., *AtomicContract cont1* in Fig. 9 binds a matched element of type 'Community' to the free variable 'COMMUNITY' such that this element can be referred to in *cont2* and *cont3*. The bindings of a set of free variables $\{var_1, \ldots, var_n\}$ (occurring in elements $\{e_1, \ldots, e_l\}$ of an *AtomicContract*) to matched elements $\{m_1, \ldots, m_n\}$ in a path condition is expressed as a binding function $l = \{(var_1, m_1), \ldots, (var_n, m_n)\}$, i.e., $l \in \mathcal{P}(FV \times BE)$, where FV and BE are the sets of free variables and bound elements, and $\mathcal{P}$ is the power set operator.

Semantics: We define a function $eval_{Atomic}(pc, c)$ that evaluates an *AtomicContract c=* (*pre,post*) for a path condition *pc* as follows:

1. If *pc* contains an isomorphic copy of *pre* but does not contain an isomorphic copy of *post*, then $eval_{Atomic}(pc, c)$ returns *false* (i.e., c does not hold for *pc* and the transformation) and an empty set of binding functions L=$\emptyset$.
2. Otherwise, $eval_{Atomic}(pc, c)$ returns *true* (i.e., c holds for *pc*) and a set of binding functions L for the free variables of c, where $L \subseteq \mathcal{P}(FV \times BE)$.

Thus, $eval_{Atomic}$ is defined as $eval_{Atomic} : PC \times AC \rightarrow \{true,\ false\} \times \mathcal{P}(FV \times BE)$, where PC is the set of path conditions of a transformation τ. Note that a set L of binding functions is returned since an *AtomicContract* may evaluate to true using different bindings of the free variables. Thus, L is constructed from all binding functions l_i returned by all possible subgraph isomorphisms.

Assuming that $FORMULAE$ is the set of elements generated by the grammar in Eqn.(1), we evaluate a formula φ for a path condition $pc \in PC$ using a function *eval:PC* × *FORMULAE* → $\{true,\ false\} \times \mathcal{P}(FV \times BE)$ as follows:

$$eval(pc, \varphi) = \begin{cases} (res_1, L_1) & \text{if } \varphi \in AC, eval_{Atomic}(pc, \varphi) = (res_1, L_1) \\ (\neg res_1, L_1) & \text{if } \varphi = \neg_{tc}\psi, eval(pc, \psi) = (res_1, L_1) \\ ((res_1 \vee res_2) \wedge C(L_1, L_2), & \text{if } \varphi = \psi \vee_{tc} \phi, eval(pc, \psi) = (res_1, L_1), \\ \qquad\qquad\qquad L_1 \cup L_2) & eval(pc, \phi) = (res_2, L_2) \\ ((res_1 \wedge res_2) \wedge C(L_1, L_2), & \text{if } \varphi = \psi \wedge_{tc} \phi, eval(pc, \psi) = (res_1, L_1), \\ \qquad\qquad\qquad L_1 \cup L_2) & eval(pc, \phi) = (res_2, L_2) \\ ((res_1 \Longrightarrow res_2) \wedge C(L_1, L_2), & \text{if } \varphi = \psi \Longrightarrow_{tc} \phi, eval(pc, \psi) = (res_1, L_1), \\ \qquad\qquad\qquad L_1 \cup L_2) & eval(pc, \phi) = (res_2, L_2) \end{cases} \tag{2}$$

where the semantics of the propositional operators ($\neg, \vee, \wedge, \Longrightarrow$) is standard, and $res_i \in \{true,\ false\}$. The consistency function $C : \mathcal{P}(FV \times BE) \times \mathcal{P}(FV \times BE) \rightarrow \{true,\ false\}$ checks for two sets of binding functions (e.g., L and L') that all free variables bound by a binding function in the first set L will always be bound to

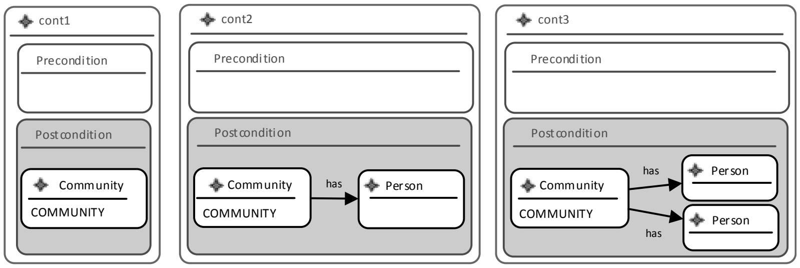

Fig. 9. Three *AtomicContracts* that can be used with different propositional operators to convey different properties for the *Persons* transformation.

the same elements by a binding function of the second set *L'* as follows:

$$\begin{aligned} C(L,L') = & \forall l \in L, \exists l' \in L' : \big(\forall v \in FV_l : ((v,m) \in l \wedge (v,m') \in l') \implies m = m'\big) \text{ and} \\ & \forall l' \in L', \exists l \in L : \big(\forall v \in FV_{l'} : ((v,m') \in l' \wedge (v,m) \in l) \implies m' = m\big) \end{aligned} \tag{3}$$

where $m, m' \in BE$, and FV_l, $FV_{l'}$ are the sets of free variables used in l and *l'* respectively. Based on the former definitions, we evaluate a formula φ for a transformation τ (with path conditions PC) using a function $eval(\tau, \varphi)$:

$$eval(\tau,\varphi) = \begin{cases} true & \text{if } \forall pc \in PC : eval(pc,\varphi) = \textit{(true,L)} \\ false & \text{otherwise} \end{cases} \tag{4}$$

where L is any set of binding functions. Thus, $eval(\tau, \varphi)$ renders a property φ to be *true* or *false* for a transformation τ by verifying φ for each path condition. Function $eval(\tau, \varphi)$ returns *true* only if for all path conditions of τ, φ holds and the bindings of all free variables consistently refer to the same elements.

Formulae of *AtomicContracts*: The new syntax and semantics allows us to formulate complex properties by composing propositional formulae of *AtomicContracts*. We demonstrate how the *AtomicContracts* in Fig. 9 (i.e., *cont1, cont2, cont3*) together with free variables can be used with different propositional operators to convey multiplicity invariants[1]. A property that mandates that the *Persons* transformation will always generate an output where every community has one or more 'Persons' (i.e., a multiplicity invariant of '1..*') can be expressed as '$cont1 \implies_{tc} cont2$'. In other words, if an element of type 'Community' is generated in the output, then this element must have at least one 'Person'. Whereas the property '$cont1 \implies_{tc} (cont2 \wedge_{tc} \neg_{tc} cont3)$' expresses a multiplicity invariant of '1..1' (i.e, if a 'Community' is generated in the output, then this 'Community' must have one 'Person' and not more).

6 Industrial Case Study

Previously in [18], we developed an industrial transformation that maps between subsets of a legacy metamodel for General Motors (GM) and the AUTOSAR metamodel. In that work, we focused on subsets of the metamodels that represent the deployment and interaction of software components. Later in [17], we proposed properties of interest for our GM-2-AUTOSAR transformation.

[1] Note that the three *AtomicContracts* in Fig. 9 have empty preconditions meaning that they will match on any input model.

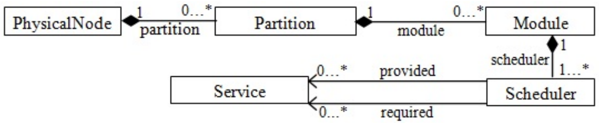

Fig. 10. Subset of the GM metamodel used by our transformation.

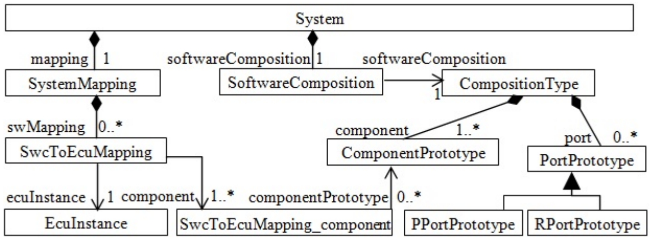

Fig. 11. Subset of the AUTOSAR metamodel used by our transformation.

We use our prover to verify the properties proposed in [17] on the GM-2-AUTOSAR transformation [18] after reimplementing it in DSLTrans. In this section, we summarize the transformation [18] and its properties [17]. Then, we discuss formulating and verifying these properties using our prover.

6.1 GM-2-AUTOSAR Model Transformation

The Source GM Metamodel: Fig. 10 illustrates the subset of the GM metamodel used in our transformation in [18][2]. A *PhysicalNode* may contain multiple *Partition*s (i.e., processing units). Multiple *Module*s can be deployed on a single *Partition*. A *Module* is an atomic, deployable, and reusable software element and can contain multiple *Scheduler*s. A *Scheduler* is the basic unit for software scheduling. It contains behavior-encapsulating entities, and is responsible for providing/requiring *Service*s to/from these behavior-encapsulating entities.

The Target AUTOSAR Metamodel: In AUTOSAR, an Electronic Component Unit (ECU) is a physical unit on which software is deployed. Fig. 11 shows the subset of the AUTOSAR metamodel [1] used by our transformation. The ECU configuration is modeled using a *System* that aggregates *SoftwareComposition* and *SystemMapping*. *SoftwareComposition* points to *CompositionType* which eliminates any nested software components in a *SoftwareComposition*. *SoftwareComposition* models the architecture of the software components (i.e., *ComponentPrototype*s) deployed on an ECU and their ports (i.e., *PPortPrototype*/ *RPortPrototype* for providing/ requiring data and services).

SystemMapping binds software components to ECUs using *SwcToEcuMappings*. *SwcToEcuMapping*s assign *SwcToEcuMapping_component*s to an *EcuInstance*. *SwcToEcuMapping_component*s, in turn, refer to *ComponentPrototype*s.

Reimplementation of the GM-2-AUTOSAR Transformation in DSLTrans: We reimplemented the GM-2-AUTOSAR transformation [18] in DSLTrans so that we can verify it in our prover. Table 1 shows the rules in each transformation layer, and the input/output types that are mapped/generated by

[2] We follow the same obfuscated naming conventions that we used for the GM metamodel in [18] for reasons of confidentiality.

Layer	Rule Name	Input Types	Output Types
1	MapPhysNode2FiveElements	PhysicalNode	System, SystemMapping, SoftwareComposition, CompositionType, EcuInstance
	MapPartition	Partition	SwcToEcuMapping
	MapModule	Module	SwCompToEcuMapping_component, ComponentPrototype
2	MapConnPhysNode2Partition	PhysicalNode, Partition	SystemMapping, EcuInstance, SwcToEcuMapping
	MapConnPartition2Module	PhysicalNode, Partition, Module	CompositionType, ComponentPrototype, SwcToEcuMapping, SwCompToEcuMapping_component
3	CreatePPortPrototype	Scheduler	PPortPrototype
	CreateRPortPrototype	Scheduler	RPortPrototype

Table 1. The rules in each layer of the GM-2-AUTOSAR transformation after reimplementing it in DSLTrans, and their input and output types.

each rule. Rules of the first and third layers create output elements. Rules of the second layer generate associations between elements created by the the first layer (shown in the actual transformation using backward links). Thus, the input and output types shown for the rules of the second layer are types that have already been matched and created and for which the rules create associations.

To represent positive application conditions (PACs) in our transformation rules, we use a combination of *Any* and *Exists* match elements (Section 2). For example, rule 'MapPhysNode2FiveElements' in Table 1 maps every *PhysicalNode* to five elements, only if the *PhysicalNode* is eventually connected to *at least* one *Module*. Thus, the *MatchModel* of rule 'MapPhysNode2FiveElements' has a *PhysicalNode* (*Any*) match element connected to *Partition* and *Module* (*Exists*) match elements. Similarly, rule 'MapModule' maps every *Module* (represented as *Any* match element) only if it is contained in one *PhysicalNode* and one *Partition* (represented as *Exists* match elements). The *MatchModel* of rule 'MapPartition' also has a *Partition* (*Any*) match element connected to *PhysicalNode* and *Module* (*Exists*) match elements to represent a PAC. Thus, the rules in the first layer totally overlap if we abstract from the match element types (i.e., *Any* or *Exists*). The extension explained in Section 4 combines the rules of the first layer into one path condition which simplifies property verification. Partially overlapping rules (Section 4) also occur in layer 2 of our transformation.

6.2 GM-2-AUTOSAR Model Transformation Properties

In [17], we stated that properties could be *invariants* or *contracts*. Invariants are properties defined on the target metamodel elements only, while contracts relate source and target metamodel elements. Based on these definitions, we further defined four categories of properties in [17]: *Multiplicity Invariants*, *Uniqueness Contracts*, *Security Invariants*, and *Pattern Contracts*. For each category, we formulated several properties that are summarized in Table 2 and discussed in [17]. We omit Uniqueness Contracts in this study since they require reasoning about attribute values, which is not yet implemented in our property prover.

Multiplicity invariants ensure that the transformation's output preserves the multiplicities in the AUTOSAR metamodel. The security invariant mandates that a *System* does not refer to a *ComponentPrototype* that is not allocated in that *System*. Pattern contracts require that if a pattern of elements is found in

Multiplicity Invariants: *(Properties defined on the target metamodel elements only)*

- ($M1$) Each *CompositionType* is associated to at least one *ComponentPrototype.*
- ($M2$) Each *SoftwareComposition* is associated to one *CompositionType.*
- ($M3$) Each *SwcToEcuMapping* is associated to at least one *SwcToEcuMapping_component.*
- ($M4$) Each *SwcToEcuMapping* is associated to one *EcuInstance.*
- ($M5$) Each *System* is associated to one *SoftwareComposition.*
- ($M6$) Each *System* is associated to one *SystemMapping.*

Security Invariant: *(Property defined on the target metamodel elements only)*

- ($S1$) All the composite *SwcToEcuMapping*s of a *System* must refer to *ComponentPrototype*s that are contained within the *CompositionType* lying under the same *System.*

Pattern Contracts: *(Properties that relate source and target metamodel elements)*

- ($P1$) If a *PhysicalNode* is connected to a *Service* through the *provided* association (in the input), then the corresponding *CompositionType* will be connected to a *PPortPrototype* (in the output).
- ($P2$) If a *PhysicalNode* is connected to a *Service* through the *required* association (in the input), then the corresponding *CompositionType* will be connected to a *RPortPrototype* (in the output).

Table 2. Properties of interest for the GM-2-AUTOSAR transformation.

the input, then a corresponding pattern of elements must be found in the output.

6.3 Verifying Properties of the GM-2-AUTOSAR Transformation

We demonstrate the formulation of pattern contracts (e.g., *P1* and P2 in Table 2) in our prover by showing the formulation of *P1* in Fig. 12 as an example. *P1* mandates that if a *PhysicalNode* is connected to a *Service* through the *provided* association in the input (as in the precondition of Fig. 12), then the corresponding *CompositionType* will be connected to a *PPortPrototype* in the output (as in the postcondition). As explained in Section 2, using a traceability link in Fig. 12 mandates that *P1* will only match *CompositionType*s that were previously created from *PhysicalNode*s. We demonstrate the formulation of '1..1' multiplicity invariants (e.g., *M2*, *M4*, *M5*, *M6*) by showing *M6* as an example. *M6* ensures that if a *System* is created in the output, then this *System* must be connected to one *SystemMapping* (and not more). Using the *AtomicContracts* in Fig. 13, *M6* can be expressed as $AC2 \Longrightarrow_{tc} (AC3 \wedge_{tc} \neg_{tc} AC4)$. Variable 'SYSTEM' mandates that if *AC2* holds for a specific *System*, then *AC3* should hold and *AC4* should not hold for the same *System.* Changing the former formula to $AC2 \Longrightarrow_{tc} AC3$ expresses a '1..*' multiplicity invariant (e.g., *M1, M3*). Using the *AtomicContracts* in Fig. 14, the security invariant *S1* can be expressed as $AC5 \Longrightarrow_{tc} AC6$. Variables 'SYSTEM' and 'COMPONENTPROTOTYPE' mandate that if *AC5* holds for a specific *System* and *ComponentPrototype* then *AC6* should also hold for the same *System* and *ComponentPrototype.*

Verification Results: We used our prover to verify the formulated properties of Table 2. The transformation was found to violate *M1* and *M3*, i.e., our prover uncovered the same bugs that we found using another tool in [17]. After examining the generated counter examples (not shown due to space limitations), we identified and fixed the two bugs in the transformation. The properties were reverified on the updated transformation, and they all returned *true.* This implies that our transformation will always satisfy the properties in Table 2.

To assess our prover's performance, we measured the time taken to generate path conditions and to verify the properties (Table 2) of the GM-2-AUTOSAR

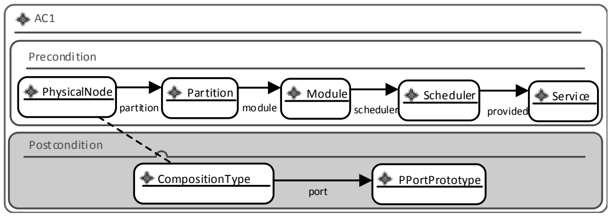

Fig. 12. One *AtomicContract* that is used to express property *P1*.

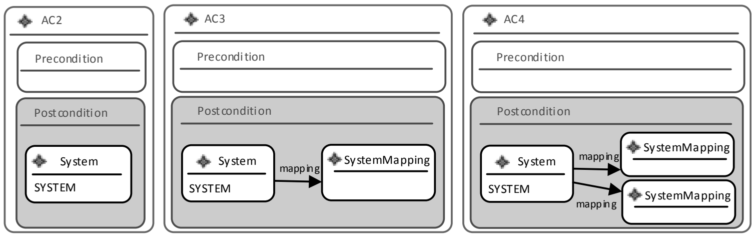

Fig. 13. Three *AtomicContracts* that are used to express property *M6*.

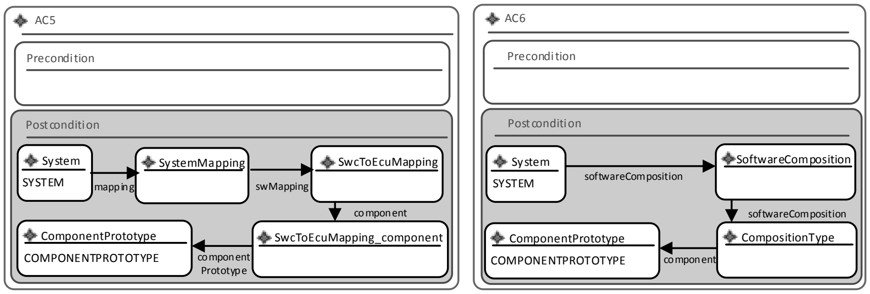

Fig. 14. Two *AtomicContracts* that are used to express property *S1*.

transformation after fixing the bugs. The prover took on average 0.6 seconds to generate the path conditions. Table 3 (first row) shows the time taken (in seconds) to verify the properties in Table 2 using the generated path conditions. We do not include the time taken for path condition generation in Table 3 since it is performed once for the transformation. The longest time taken to verify a property was 0.02 seconds (*P1*, *P2*). Thus, our prover can verify an industrial transformation's properties in a short time. More experiments are needed before we can claim that our prover scales to transformations of varying complexities.

Our property prover and the transformation used in [14] is available at [13]. The industrial transformation is not included for confidentiality reasons.

7 Discussion

We discuss the strengths and limitations of our prover by comparing it to a tool that we used to verify the GM-2-AUTOSAR transformation in [17]. The tool we used in [17] verifies ATL (textual) transformations by translating them to a relational representation and then using constraint solvers to prove properties for the translated transformation within a scope (i.e., maximum number of objects per class). In contrast, the prover described in this study verifies DSLTrans

Property	M1	M2	M3	M4	M5	M6	S1	P1	P2
Verification Time (our property prover)	.013	.017	.013	.017	.017	.019	.017	.02	.02
Verification Time ([17] at scope 6)	76	73.4	75	75	75.5	74.5	114	256	251

Table 3. Time taken (in seconds) to verify the properties in Table 2 using our property prover (first row) and using a tool based on constraint solving [17] (second row).

(graphical) transformations in their native form (i.e., without translating them to another formalism) using the symbolic transformation executions.

We identify three strengths of our prover in comparison with the tool we used in [17]. First, our prover's verification result holds for all transformation executions and is not limited to a scope. Second, our prover verifies the transformation without translating it to another formalism. Third, our prover verified the properties faster than the tool we used in [17]. Table 3 shows the time taken to verify the properties in Table 2 using our prover (first row) and using the tool in [17] (second row). In Table 3, we only show the results for the smallest scope we used in [17] (i.e., 6). As shown in Table 3, our prover takes significantly shorter time to exhaustively verify the properties, whereas much longer times were needed to verify the same properties in a scope of 6 in [17]. Thus, we claim that our prover scales well in comparison with the tool we used in [17].

We identify two limitations of our prover in comparison with the tool we used in [17]. First, although negative application conditions (NACs) are expressible in DSLTrans, our prover cannot verify transformations with rules having NACs. Second, our prover cannot verify properties that reason about attribute values such as the uniqueness contracts (Section 6.2) that we were able to verify in [17]. We are currently working on addressing both limitations in our prover.

8 Related Work

We review studies that propose (1) verification techniques and tools that are input-independent and (2) property languages similar to ours.

(1) Büttner et al. [9] and Cabot et al. [10] translated a transformation and its metamodels into a transformation model and used model finders (e.g., UML2Alloy) and constraint solvers (e.g., UMLtoCSP) to verify a transformation property. Anastasakis et al. [3] and Baresi and Spoltini [6] translated a transformation and its metamodels into an Alloy model and used the Alloy Analyser to verify the Alloy model within a scope. Troya and Vallecillo [20] translated a transformation into Maude and used Maude's analysis capabilities to verify the transformation. Becker et al. [8] verified if a transformation can generate *forbidden* patterns by checking if the backward application of each rule to each forbidden pattern can produce a valid input. Orejas and Wirsing [15] proposed translating graphs to triple algebras to verify (e.g., using Maude) propositional formula of properties. The study claimed that verifying graph transformations is difficult, and hence the need for the translation to algebra. Approaches used by tools such as Henshin [4] and AGG [19] are input-dependent.

(2) Büttner et al. [9] expressed properties in OCL and verified them using model finders. Guerra et al. [11] proposed PaMoMo, a graphical language

to express contracts and complex properties that manipulate contracts. These properties can be compiled into OCL and injected into any OMG-based transformation implementation (e.g., ATL) for automated verification. Asztalos et al. [5] formulated properties and rules as assertions in first-order logic. Deduction rules were then used to deduce the property assertion from the rules' assertions. AGG's [19] property language is similar to ours (i.e., contracts that can be used to build complex properties) except that AGG's verification is input-dependent.

Difference between our study and related work: Our study differs from the surveyed studies in one or more of the following aspects. First, verification is performed on an intuitive, graphical language that does not require a mathematical background to be used, e.g., Maude [20, 15]. Second, we used our prover to verify a simple and an industrial transformation. Third, we demonstrated several property kinds that our prover can verify as opposed to verifying specific properties, e.g., forbidden patterns [8]. Fourth, verification is based on generating the symbolic executions. Fifth, we have proved the *soundness* and *completeness* of our technique in [12]. Many studies translated a transformation into another formalism and verified properties on the translated transformation [9, 10, 3, 6, 20, 15]. Such approaches should prove the soundness of the translated transformation before verifying properties. Moreover, such approaches should translate the verification result back to the original formalism for comprehension. Other studies proposed incomplete techniques that are restricted to a scope [9] or that do not guarantee that the transformation is fault-free, e.g., testing.

While textual property languages (e.g., OCL [9] and assertions [5]) have been used for specifying properties, we believe that a graphical property language is useful as more researchers adopt graph transformations due to their intuitive, graphical format. Approaches where graphical properties are translated into a textual formalism (e.g., [11]) have two drawbacks. First, the soundness of the translation should be proved before verifying the translated properties. Second, the translated properties in [11] can not be used to automatically verify transformations implemented in graph-based transformation languages.

We believe that our graph-based property language (that can be verified without translation to another formalism) and input-independent verification technique advances the state of the art and may encourage users in safety critical domains to use the more intuitive, graph-based transformation languages.

9 Conclusion and Future Work

In this study we extended a symbolic model transformation property prover [14, 12] that initially only verified *AtomicContracts*. The extended prover now verifies *AtomicContracts* and propositional formulae of *AtomicContracts* for DSLTrans transformations. We have also extended the path condition generation algorithm presented in [12] by treating overlapping rules. Further, we demonstrated our property prover on an industrial case study [18]. We showed that the prover is of practical use and features fast property proving times when compared with another prover. We also discussed the strengths and limitations of our prover.

For future work, more experiments on bigger transformations are needed to test the prover's scalability. Moreover, as mentioned in Sections 6.2 and 7, we plan to handle NACs and attribute values when generating the set of path conditions to facilitate verifying properties that reason about attribute values.

References

1. AUTOSAR Consortium. AUTOSAR System Template, http://AUTOSAR.org/index.php?p=3&up=1&uup=3& uuup=3&uuuup=0& uuuuup=0/AUTOSAR_TPS_SystemTemplate.pdf, 2007.
2. M. Amrani, L. Lúcio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J. R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *VOLT*, pages 921–928, 2012.
3. K. Anastasakis, B. Bordbar, and J. Küster. Analysis of Model Transformations via Alloy. *MoDeVVa*, pages 47–56, 2007.
4. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *MoDELS*, pages 121–135. Springer, 2010.
5. M. Asztalos, L. Lengyel, and T. Levendovszky. Towards Automated, Formal Verification of Model Transformations. In *ICST*, pages 15–24, 2010.
6. L. Baresi and P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *ICGT*, volume 4178 of *LNCS*, pages 306–320, 2006.
7. B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *SLE*, pages 296–305. 2011.
8. B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *ICSE*, 2006.
9. F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL Transformations Using Transformation Models and Model Finders. In *ICFEM*, volume 7635 of *LNCS*, pages 198–213, 2012.
10. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and Validation of Declarative Model-to-Model Transformations Through Invariants. *Systems and Software*, 83(2):283–302, 2010.
11. E. Guerra, J. de Lara, D. Kolovos, and R. Paige. A Visual Specification Language for Model-to-Model Transformations. In *VL/HCC*, pages 119–126. IEEE, 2010.
12. L. Lúcio, B. Oakes, and H. Vangheluwe. A Technique for Symbolically Verifying Properties of Graph-Based Model Transformations. Technical Report SOCS-TR-2014.1, McGill U., 2014.
13. L. Lúcio and G. Selim. DSLTrans Property Prover and Example Transformation, http://msdl.cs.mcgill.ca/people/levi/police_station_verification_example.zip.
14. L. Lúcio and H. Vangheluwe. Model Transformations to Verify Model Transformations. In *VOLT*, 2013.
15. F. Orejas and M. Wirsing. On the Specification and Verification of Model Transformations. In *Semantics and algebraic specification*, pages 140–161. Springer, 2009.
16. L. A. Rahim and J. Whittle. A Survey of Approaches for Verifying Model Transformations. *SoSyM*, pages 1–26, 2013.
17. G. Selim, F. Büttner, J. R. Cordy, J. Dingel, and S. Wang. Automated Verification of Model Transformations in the Automotive Industry. In *MODELS*, 2013.
18. G. Selim, S. Wang, J. R. Cordy, and J. Dingel. Model Transformations for Migrating Legacy Models: An Industrial Case Study. *ECMFA*, pages 90–101, 2012.
19. G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE*, pages 446–453. Springer, 2004.
20. J. Troya and A. Vallecillo. A Rewriting Logic Semantics for ATL. *JOT*, 10:5: 1–29, 2011.

Appendix - Overlapping Rules

This appendix elaborates on the generation of path conditions when overlapping rules occur in a transformation (explained at the end of Section 4).

The shape of our industrial transformation introduced the need to treat rules that have overlapping MatchModels in path condition generation. Overlapping rules are defined as follows: for two rules in the same layer, the MatchModel of one rule syntactically *subsumes* (contains) the MatchModel of the other rule. More concretely and formally, we mean that there is an injective graph homomorphism between the match elements of one rule and the other rule that respects the metamodel classes of the match elements, as well as the types of the relations between those elements (but is independent of whether those classes are of type *Any* or *Exists*). Overlapping rules require specific handling during path condition generation. Intuitively, when two rules overlap then the execution of the 'larger' rule necessarily implies the execution of the 'smaller' rule as they match over the same set of input elements.

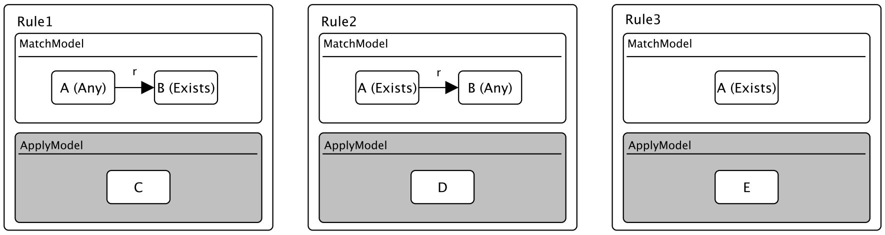

Fig. 15. Three overlapping rules.

Given two rules R and R', for our path condition generation purposes we can distinguish two types of rule overlap:

1. **Total Overlap:** This case happens when R *subsumes* R' and R' *subsumes* R. In Fig. 15, *Rule1* and *Rule2 totally overlap*.
2. **Partial Overlap:** This case happens when R *subsumes* R' but not the other way around. In Fig. 15, rules *Rule1* and *Rule2 subsume Rule3*, but *Rule3* does not subsume *Rule1* or *Rule2*. As such, *Rule1* and *Rule3*, as well as *Rule2* and *Rule3*, *partially overlap*.

In path condition generation, the above two cases are treated differently. In order to treat case 1 where two or more rules *totally overlap*, a step preceding the path condition generation algorithm is executed that merges the totally overlapping rules into one rule. This new merged rule contains the MatchModel that is shared by both rules, as well as the disjoint union of the ApplyModels of the two rules. This allows us to symbolically represent the fact that the two rules always execute together. We present in Fig. 16 the result of merging *Rule1* and *Rule2*, as well as the result of merging *Rule2* and *Rule3*[3]. The fact that the match elements are of type *Any* or *Exists* plays a role in the merge: when two

[3] We use the notation $[R\,R']$ to refer to the result of merging rules R and R'. Note that the *merge* operation is n-ary.

match elements m and m' are merged, the resulting element will be of type *Any* if either m or m' are *Any*. The resulting element will be of type *Exists* only if both m and m' are of type *Exists*[4].

Finally, the original rules participating in the merge are removed from the layer and the merged rule is added.

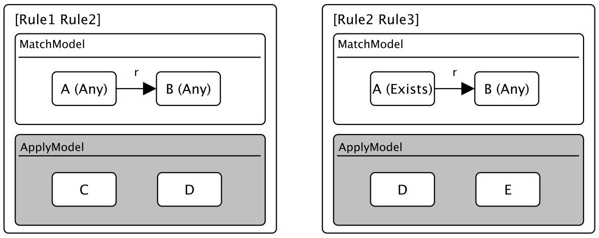

Fig. 16. Rule merge examples.

In order to treat case 2 where rules *partially overlap*, additional reasoning during path condition generation is necessary. In [12], we have argued that in order to build a set of path conditions for a single layer, all combinations of rules need to considered (i.e., the power set of the set of rules in the layer is calculated). However, assume that a layer includes four rules, R_1, R_2, R_3 and R_4. Assume further that the *subsumes* relation between those rules induces the partial order in Fig. 17. Since R_2 subsumes R_3 (noted in what follows $R_2 \geq R_3$), if rule R_2 executes, then R_3 also necessarily executes and they can be merged as shown in Fig. 16. Similarly, if R_1 executes, then all the rules that R_1 subsumes necessarily execute and can be merged with R_1. Rules R_3 and R_4 can execute on their own or combined, since they do not subsume any other rules. Finally, rule R_4 can also execute with or without the merge of R_2 and R_3 given they belong to different branches of the partial order in Fig. 17.

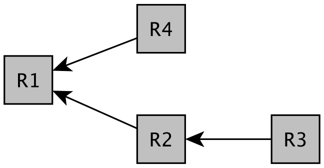

Fig. 17. A partial order between rules induced by MatchModel subsumption.

[4] During rule merge, *Any* match elements take precedence over *Exists* match elements. This is so because, symbolically, the merged match element represents an arbitrarily large number of matches of that element in an input model (from the *Any* match element), plus one match (from the Exists element). It is relevant to keep track of whether the match elements are *Any* or *Exists* because this allows to reason about the number of instances produced by the transformation, although this research is beyond the scope of this paper.

Thus, for the case where rules partially overlap within a layer, not all rule combinations of that layer are meaningful for path condition generation. This is because the execution of some rules necessarily implies the execution of others and thus certain rules cannot execute in isolation. The algorithm we describe next takes as input the set of rules within a layer and the partial order induced by the subsumption relation between those rules (such as the one shown in Fig. 17), if any. The algorithm returns the set of rules and merged rule combinations that should be considered for path condition generation for that layer. The algorithm starts by building the set of all possible rule combinations, independently of partial overlaps. Then, the algorithm eliminates from that set any rule combinations that cannot occur because they include rules that cannot execute in isolation. Finally, the algorithm merges rules that must execute together within a combination. The algorithm is as follows:

1. Generate the power set of the rules in the layer. For our example in Fig. 17, this step produces the set $\big\{\emptyset, \{R_1\}, \{R_2\}, \{R_3\}, \{R_4\}, \{R_1, R_2\}, \{R_1, R_3\}, \{R_1, R_4\}, \{R_2, R_3\}, \{R_2, R_4\}, \{R_3, R_4\}, \{R_1, R_2, R_3\}, \{R_1, R_2, R_4\}, \{R_1, R_3, R_4\}, \{R_2, R_3, R_4\}, \{R_1, R_2, R_3, R_4\}\big\}$, including $2^4 = 16$ rule combinations. We call this set PC, the set of path conditions for this layer.
2. Calculate the merges of each rule in the partial order with all the rules it subsumes. In our example, this step produces the merges $[R_1R_2R_3R_4]$ and $[R_2R_3]$. Intuitively, these merges correspond to sets of rules that necessarily execute together. We call the set of merged rules for this layer $M = \{[R_2R_3], [R_1R_2R_3R_4]\}$.
3. Remove a rule combination from PC if it contains a rule R and does not contain a rule R' such that $R \geq R'$. In our example, this corresponds to removing rule combinations $\{R_1\}$, $\{R_2\}$, $\{R_1, R_2\}$, $\{R_1, R_3\}$, $\{R_1, R_4\}$, $\{R_2, R_4\}$, $\{R_1, R_2, R_3\}$, $\{R_1, R_2, R_4\}$, and $\{R_1, R_3, R_4\}$. Intuitively, this step removes a rule combination from PC if that combination contains rules that cannot execute in isolation, but the set of additionally required rules does not exist in that combination.
4. Search for rule combinations of PC that contain all the rules composing one of the rule merges of M. When found, replace the elements in the rule combination by their corresponding merge. In our example this would mean replacing rule combination $\{R_1, R_2, R_3, R_4\}$ by $\{[R_1R_2R_3R_4]\}$ and rule combination $\{R_2, R_3, R_4\}$ by $\{[R_2R_3], R_4\}$. Intuitively, this step merges in the remaining rule combinations the rules that necessarily execute together.

After executing the above algorithm, the resulting set of path conditions in our example is $PC = \big\{\emptyset, \{R_3\}, \{R_4\}, \{[R_2R_3]\}, \{R_3, R_4\}, \{[R_2R_3], R_4\}, \{[R_1R_2R_3R_4]\}\big\}$, containing only 7 rule combinations. This implies a drastic reduction of more than half of the number of path conditions to be considered for verification for this layer.

Note that, unlike the case where rules totally overlap, partially overlapping rules cannot be processed by statically altering the rules of a transformation. As we have shown above, partially overlapping rules require dynamic treatment during the generation of the path conditions for each layer.