Symbolic Model Transformation Property Prover for DSLTrans

Gehan Selim, Levi Lúcio, James R. Cordy, Juergen Dingel

{gehan, cordy, dingel}@cs.queensu.ca, levi@cs.mcgill.ca

Technical Report 2013-616

School of Computing, Queen's University

Kingston, Ontario, Canada, K7L 2N8

December 2013

©2013 Gehan Selim, Levi Lúcio, James R. Cordy, and Juergen Dingel

Contents

1 Introduction **1**

2 Background **1**

2.1 The DSLTrans Model Transformation Language 2

2.2 Expressing *AtomicProperties* in DSLTrans 6

3 The Symbolic Model Transformation Property Prover **8**

3.1 Generating the Set of Path Conditions for a DSLTrans Model Transformation 9

3.2 Verification Using the Symbolic Model Transformation Property Prover 10

3.2.1 Design of the Verification Technique 11

3.2.2 Verifying *AtomicProperties* 12

3.2.3 Verifying Propositional Logic Formulae of Simple Properties 15

4 Industrial Case Study **17**

4.1 GM-2-AUTOSAR Model Transformation 17

4.2 GM-2-AUTOSAR Model Transformation Properties 19

4.3 Verifying Properties of the GM-2-AUTOSAR Model Transformation 20

5 Enabling Technology **23**

6 Related Work **24**

7 Conclusion and Future Work **25**

A The Back-End Component of the Symbolic Model Transformation Property Prover **29**

List of Figures

1 Two metamodels for describing different views for the chain of command in police stations. 2

2 A model transformation expressed in DSLTrans 3

3 An example input (left) and output model (right) for the Police Station transformation 4

4 Police Station Transformation Property 1 7

5 Police Station Transformation Property 2 7

6 The complete architecture of our symbolic model transformation property prover. 8

7 A demonstration of how our property prover generates the set of path conditions for a DSLTrans transformation in iterations. 10

8 The design of the verification technique used in our symbolic model transformation property prover. 11

9 An example of two rules with common elements in their MatchModels that can be collapsed. 14

10 The result of collapsing the two rules in Fig. 9 into one rule. 15

11 Subset of the GM metamodel directly used by our transformation in [16]. 18

12 Subset of the AUTOSAR System Template directly used by our transformation. 18

13 Expressing property *M1* using two *AtomicProperties*. 22

14 The six higher order transformations (HOTs) executed by the preprocessing component of our symbolic model transformation property prover. 29

15 The abstract rule metamodel for the Police Station transformation. 31

List of Tables

1 The rules in each layer of the GM-2-AUTOSAR transformation after reimplementing it in DSLTrans, and their input and output types. . 19

2 Formulated OCL Constraints 21

3 The time taken by our symbolic model transformation property prover (in seconds) to verify each of the 9 properties shown in Table 2. 23

1 Introduction

As Model Driven Development (MDD) is finding more widespread use for developing software, *model transformations* are becoming an increasingly active research field since they can automate the entire MDD process. Model transformations map input model(s) conforming to a source metamodel into output model(s) conforming to a target metamodel. Since model transformations are intended to be repeatedly used for a class of models, model transformation verification is a crucial task in MDD. Several studies discussed different model transformation verification approaches and surveys of the proposed approaches have been conducted [17, 4].

This study investigates verifying model transformations implemented in the DSLTrans model transformation language [7]. DSLTrans is a Turing-incomplete graphical model transformation language that guarantees termination and confluence by construction. We propose a symbolic model transformation property prover for DSLTrans which is based on building the set of path conditions (i.e., possible symbolic executions) of a DSLTrans transformation and checking each of these path conditions for some property. The basic idea underlying our property prover has been previously presented in [13] and further refined in [14]. We describe the additions made to our symbolic model transformation property prover to facilitate proving both atomic properties (i.e., constraints on relations between input and output models) and propositional logic formulae that manipulate such atomic properties. We discuss the different steps carried out by our symbolic model transformation property prover on a simple model transformation as a running example. Further, we demonstrate the application of our property prover on a more complicated industrial model transformation that we previously reported on in [16].

The rest of this paper is organized as follows: Section 2 summarizes the basic concepts of DSLTrans and the simplest properties that can be expressed in it; Section 3 discusses in detail how the property prover generates the set of path conditions for a DSLTrans transformation and the verification technique it uses to verify properties of varying complexities; Section 4 demonstrates the application of our property prover to an industrial case study and the obtained results ; Section 5 summarizes the underlying technologies used to build our property prover; Section 6 reviews similar studies in the literature; and Section 7 concludes our study and presents future work.

2 Background

Barroca et al. [7] previously proposed the DSLTrans model transformation language. In another study, Lúcio et al. [13] demonstrated the formulation and verification

of simple, atomic properties in DSLTrans. In this section, we summarize the basic concepts of DSLTrans [7] and the atomic properties that can be expressed in the language, as proposed in [13]. Later in Section 3, we demonstrate how the verification technique discussed in [13] was improved to facilitate expressing and verifying the simple properties described in [13] and more complicated properties.

2.1 The DSLTrans Model Transformation Language

The DSLTrans model transformation language is a graph-based model transformation language that can be used to specify exogenous transformations that are confluent and terminating by construction. DSLTrans is non-Turing complete since it does not have constructs that imply unbounded recursion or non-determinism. In what follows, we demonstrate DSLTrans and its constructs using a simple DSLTrans transformation as a running example.

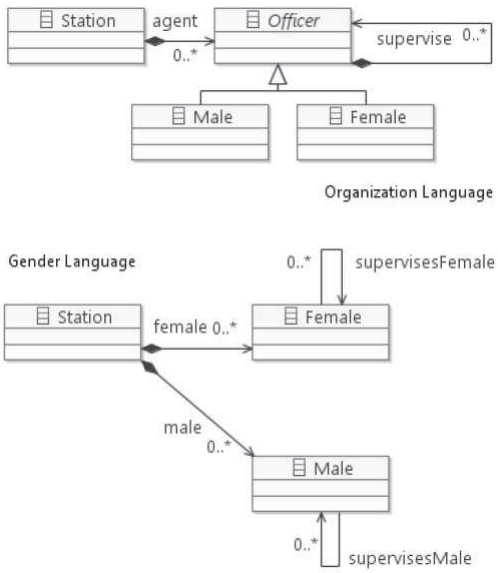

Figure 1: Two metamodels for describing different views for the chain of command in police stations.

Fig. 1 presents two metamodels of languages used to describe different views of a police station organization . The 'Organization Language' metamodel represents a language for describing the chain of command in a police station in which *Male* and *Female* officers can supervise each other. The 'Gender Language' metamodel represents a language for describing a different view of a police station's chain of command, where the officers working at the police station are classified by gender and only officers of the same gender can supervise each other.

Fig. 2 demonstrates a transformation between models of both languages

implemented in DSLTrans[1]. The purpose of this transformation is to flatten a chain of command given in the 'Organization Language' metamodel into two independent sets of male and female officers in the 'Gender Language' metamodel. Within each of these sets of male and female officers, the command relations are kept, i.e., a female officer will be directly related to all her female subordinates and likewise for the male officers. In the rest of this paper, we will refer to the transformation shown in Fig. 2 as the *Police Station* transformation.

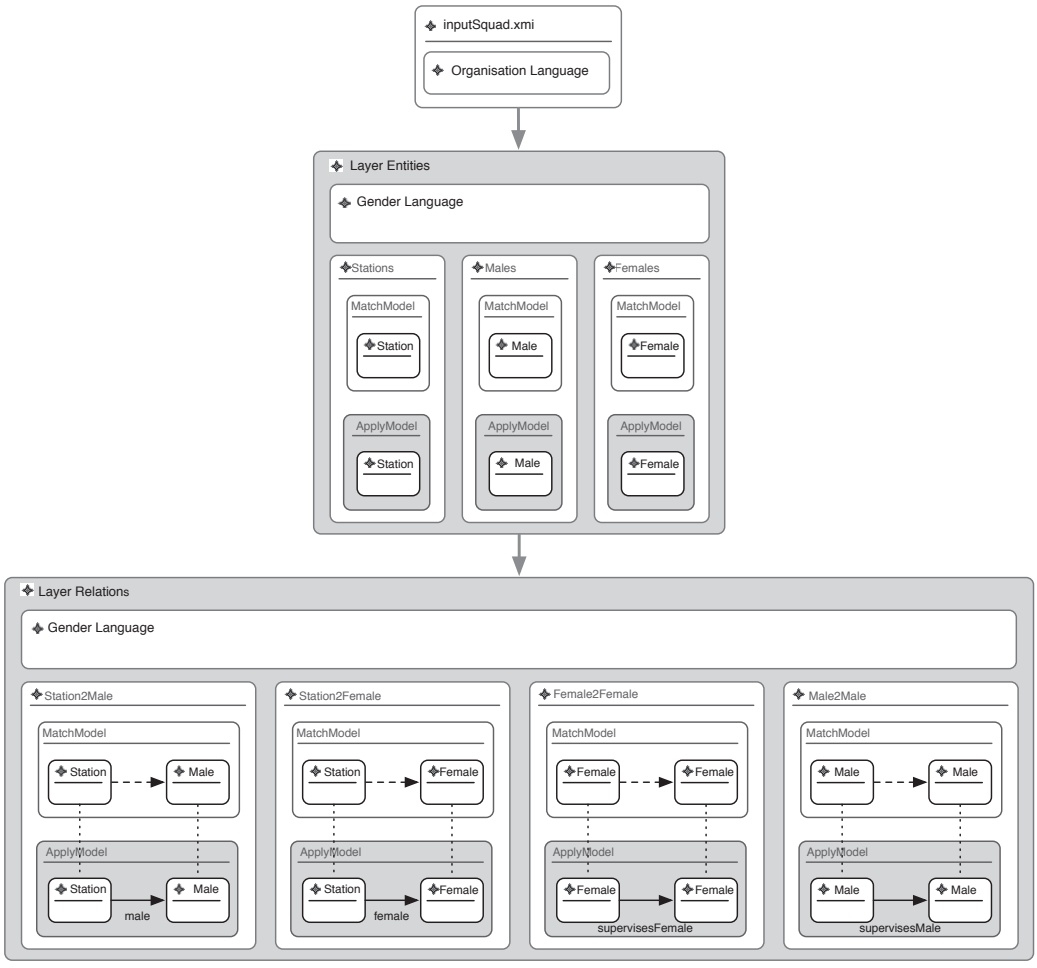

Figure 2: A model transformation expressed in DSLTrans

Fig. 3 demonstrates an example of an input model for the Police Station transformation (on the left) and its corresponding output model (on the right). Notice that the elements s, m_k and f_k on the left of Fig. 3 are instances of the source metamodel classes *Station*, *Male* and *Female* (i.e., the 'Organization Language'

[1]DSLTrans has been implemented as an Eclipse plug-in [1] and the example shown in Fig. 2 is expressed using the implemented plug-in.

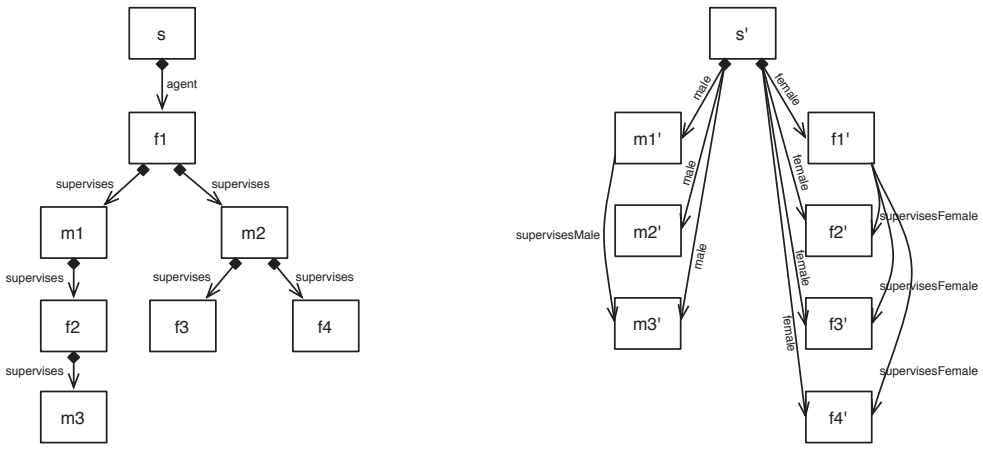

Figure 3: An example input (left) and output model (right) for the Police Station transformation

metamodel in Fig. 1). The primed elements on the right of Fig. 3 are their corresponding instances from the target metamodel (i.e., the 'Gender Language' metamodel in Fig. 1).

We can identify several components in the Police Station transformation shown in Fig. 2. A DSLTrans transformation is composed of a set of input model sources called file-ports (e.g., inputSquad.xmi in Fig. 2) and a set of layers (e.g. 'Entities' and 'Relations' layers in Fig. 2). File-ports and layers are typed according to metamodels. DSLTrans sequentially executes the layers of a transformation specification. A layer is composed of transformation rules that execute in a non-deterministic order but produce a deterministic result. Each transformation rule is a pair (*MatchModel*, *ApplyModel*) where *MatchModel* is a pattern of source metamodel elements and *ApplyModel* is a pattern of target metamodel elements. For example, the MatchModel of the transformation rule 'Stations' in the 'Entities' layer (Fig. 2) holds one 'Station' class from the 'Organization Language' metamodel (i.e., the source metamodel) and the ApplyModel holds one 'Station' class from the 'Gender Language' metamodel (i.e., the target metamodel). This means that all input model elements of type 'Station' (of the 'Organization Language' metamodel) will be transformed into output model elements of type 'Station' (of the 'Gender Language' metamodel).

Several DSLTrans constructs can be used to build the MatchModel of a DSLTrans transformation rule, as shown in the Police Station transformation (Fig. 2).

- *Match Elements* are variables typed by classes of the source metamodel that can assume as values instances of that class (or subclass) from the input

model. An example of a match element is the 'Station' element in the MatchModel of the 'Stations' rule in the 'Entities' layer.

- A match element can be either an *AnyMatchClass* or an *ExistsMatchClass*. We demonstrate the difference between these two constructs in the following example. Using an *AnyMatchClass* of type 'Station' in the MatchModel of the 'Stations' rule in Fig. 2 implies that the rule will create an output 'Station' object (of the 'Gender Language' metamodel) *for every* 'Station' object (of the 'Organization Language' metamodel) matched in the input model. Alternatively, using an *ExistsMatchClass* of type 'Station' in the MatchModel of the 'Stations' rule in Fig. 2 implies that the rule will create only one output 'Station' object (of the 'Gender Language' metamodel) if at least one 'Station' object (of the 'Organization Language' metamodel) was matched in the input model. In our property prover, the *cardinality* attribute of a match element is used to differentiate between an *AnyMatchClass* and an *ExistsMatchClass*. An *AnyMatchClass* is a match element with a cardinality of '+', while an *ExistsMatchClass* is a match element with a cardinality of '1'. For the Police Station transformation shown in Fig. 2, all the match elements are *AnyMatchClass*es.

- *Attribute Conditions* are conditions over the attributes of a match element (i.e., an *AnyMatchClass* or an *ExistsMatchClass*).

- *Direct Match Links* are links between two match elements in a rule's MatchModel that are typed by labelled relations of the source metamodel. These variables can assume as values relations having the same label in the input model.

- *Indirect Match Links* are similar to direct match links, but there may exist a path of containment associations between the linked match elements. Our notion of indirect match links captures only acyclic EMF containment associations to avoid cycles and infinite amounts of matches over the transitive closure of associations in the input model. In Fig. 2, indirect match links appear in all the rules of the 'Relations' layer as horizontal, dashed arrows between match elements.

- *Backward Links* connect elements of the MatchModel and the ApplyModel of a certain rule. Backward links are used in all the transformation rules in the 'Relations' layer (Fig. 2) and are annotated as vertical, dashed lines. Backward links are used to refer to elements created in a previous layer (i.e., to distinguish them from elements to be created from scratch). Thus, the only possibility to reuse elements created from a previous layer is to refer to them using backward links;

- *Negative Conditions* are used to express undesirable conditions over match elements, backward links, direct match links, and indirect match links.

Similar DSLTrans constructs can be used to build the ApplyModel of a DSLTrans transformation rule, as shown in the Police Station transformation (Fig. 2).

- *Apply Elements and Apply Links*: Similar to match elements, apply elements are variables typed by classes of the target metamodel. Apply elements in a transformation rule that are not connected to backward links will create elements of the same type in the output model for each time the MatchModel is found in the input model. A similar mechanism is used for apply links. Apply elements that are connected to backward links are handled in a different way. For example, the 'Station2Male' transformation rule of the 'Relations' layer takes instances of 'Station' and 'Male' (of the 'Gender Language' metamodel) that were created in a previous layer from instances of 'Station' and 'Male' (of the 'Organization Language' metamodel), and connects them using a 'male' relation.
- *Apply Attributes*: DSLTrans includes a relatively small language for setting the attribute values of apply elements from references to one or more attributes of match elements.

More details on the syntax and semantics of DSLTrans can be found in [7].

2.2 Expressing *AtomicProperties* in DSLTrans

AtomicProperties are the simplest properties that can be expressed and verified using our symbolic model transformation property prover. They specify a constraint of the following form: "if a structural relation between some input model elements holds, then another structural relation between some output model elements should also hold". Thus, an *AtomicProperty* is composed of a *precondition* and a *postcondition*. A precondition specifies a constraint on the transformation's input model in the form of a structural relation between input model elements. Similarly, a postcondition specifies a constraint on the transformation's output model in the form of a structural relation between output model elements.

Preconditions use the same constructs used in the MatchModel of DSLTrans transformation rules, including the possibility of expressing several occurrences of the same metamodel element and indirect links. Indirect links in properties have the same meaning as in the MatchModel of DSLTrans transformation rules; they involve patterns over the transitive closure of containment links in input models.

Postconditions also use the same constructs as the ApplyModel of DSLTrans transformation rules, with the additional possibility of expressing indirect links for patterns involving the transitive closure of containment links in output models. A formal definition of our property language can be found in [13].

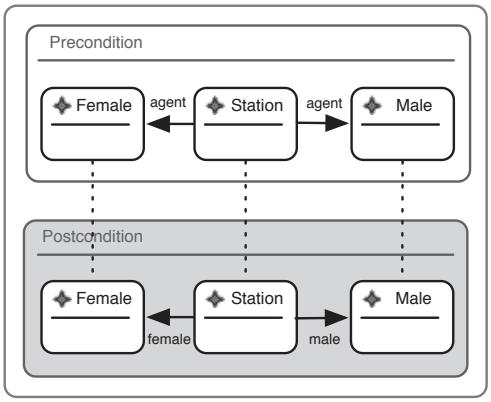

Figure 4: Police Station Transformation Property 1

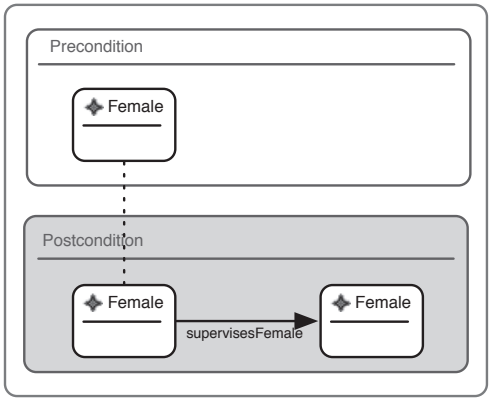

Figure 5: Police Station Transformation Property 2

Fig. 4 and Fig. 5 demonstrate two properties that we wish to verify for all possible executions of the Police Station transformation shown in Fig. 2. The property in Fig. 4 is interpreted as follows: "A model which includes a police station that has both a male and a female officers as *agents* will be transformed into a model where the male officer will exist in the *male* set and the female officer will exist in the *female* set". Intuitively, we expect that the property shown in Fig. 4 will hold for all transformation executions. The property in Fig. 5 is interpreted as follows: "Any model which includes female officer will be transformed into a model where that female officer will always supervise another female officer". We do not expect that the property shown in Fig. 5 will hold for all transformation executions.

The goal of our symbolic model transformation property prover is to verify that a property such as the one shown in Fig. 4 will hold for the Police Station transformation when run on any input model and that a property such as the one shown in Fig. 5 will not necessarily hold for the Police Station transformation when run on any input model.

3 The Symbolic Model Transformation Property Prover

The basics of our symbolic model transformation property prover have been laid out in [13] and further refined in [14]. Fig. 6 demonstrates the final architecture of our property prover after adding several aspects to what was presented in [13, 14]. In general, our symbolic model transformation property prover takes four inputs: the DSLTrans model transformation of interest, the transformation's source and target metamodels, and the property to verify. Verification is then carried out as shown in the main component of our property prover in Fig. 6:

- The property prover generates the set of *path conditions* representing the possible executions of the input transformation.
- The property prover verifies the input property on the generated set of path conditions and renders the property to be either *true* or *false* for the transformation when run on any input model. Thus, according to the classification we presented in [4], the verification technique used by our property prover is *transformation-dependent* and *input-independent*.

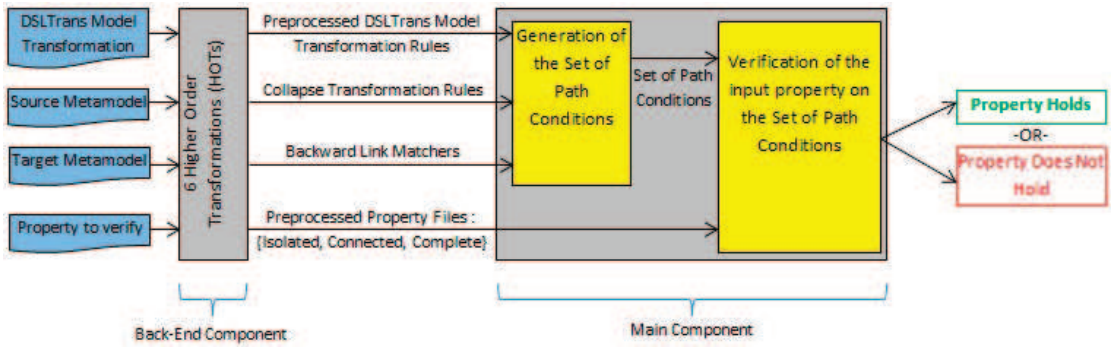

Figure 6: The complete architecture of our symbolic model transformation property prover.

To perform the two steps in the property prover's main component, a preprocessing phase is executed first by a back-end component. In this preprocessing phase,

six higher order transformations (HOTs) are executed to generate the artifacts required by the property prover's main component to verify properties of the input DSLTrans transformation.

In this section, we explain the steps carried out by the property prover's main component, i.e., we explain generating the set of path conditions and verifying properties using the generated set of path conditions. In the relevant parts of this section, we point out and define the artifacts generated by the property prover's back-end component. The inner workings of the back-end component are summarized in Appendix A and discussed in detail in [14].

3.1 Generating the Set of Path Conditions for a DSLTrans Model Transformation

The main component of our symbolic model transformation property prover generates a set of *path conditions* on which the input property will be verified. These path conditions represent all the possible executions of the transformation under verification, i.e., the set of possible input patterns that will trigger the transformation and their corresponding set of output patterns.

The set of path conditions is generated using the preprocessed transformation rules (produced by the prover's back-end component as shown in Fig. 6) after adding to each rule *traceability links* between each element in the rule's MatchModel and each element in the rule's ApplyModel. *Traceability links* are used to keep track of which output model elements came from which input model elements[2].

Using the preprocessed transformation rules (after adding *traceability links* to them), our property prover builds the set of path conditions over iterations. More precisely, for a DSLTrans transformation with n layers, we generate the set of path conditions in *(n+1)* iterations. In Fig. 7, we roughly demonstrate how our property prover generates the set of path conditions for the Police Station transformation (Fig. 2) in iterations. We identify every rule in each layer of Fig. 2 by a number with an index. For example, the index 1_1 corresponds to the first rule (ordered from left to right in Fig. 2) in the first transformation layer (i.e., the 'Entities' layer). The set of path conditions starts with the empty path condition in iteration 0, where no transformation rule has been applied. To generate path conditions in iteration 1, the empty path condition from iteration 0 connects to all possible rule combinations of the first transformation layer (i.e., the *power set* of the rules of the first transformation layer). Similarly, to generate path conditions in iteration 2, each path condition from iteration 1 then is merged with all *applicable* rule combinations of the second transformation layer (i.e., the *power set* of the rules

[2] *Traceability links* can also be used in *AtomicProperties* to link elements from the property's precondition to elements from the property's postcondition.

of the second transformation layer). By *applicable* rule combination we mean that if a specific rule combination from the second transformation layer has backward links, then this rule combination is connected to a path condition from iteration 1 if the path condition from iteration 1 generates the required elements (as per the traceability links). This check is performed using the backward link matchers generated by the property prover's back-end component, as shown in Fig. 6.

Therefore, the path conditions generated in iteration i include not only the power set of rules from the i^{th} transformation layer, but also rules from the path conditions generated until iteration *i-1*. Thus, each path condition accumulates all the preprocessed transformation rules leading to it, i.e., each path condition represents all input model patterns leading to it and their corresponding output model patterns (with possible *traceability links* between these two patterns). We refer to the accumulated MatchModels of all the rules in a path condition as the *match pattern* of the path condition. Similarly, we refer to the accumulated ApplyModels of all the rules in a path condition as the *apply pattern* of the path condition.

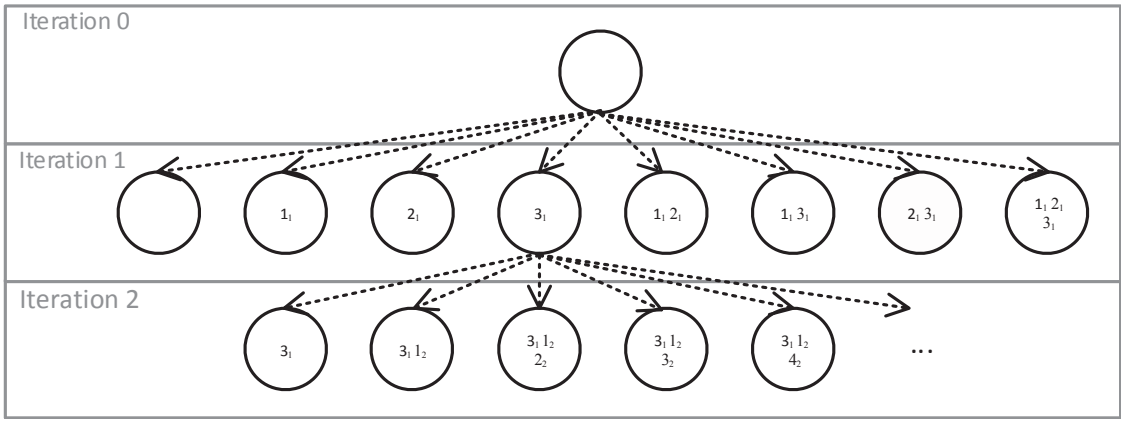

Figure 7: A demonstration of how our property prover generates the set of path conditions for a DSLTrans transformation in iterations.

Only the set of path conditions generated in the last iteration are returned as the resultant set of path conditions since they capture all the possible *complete* transformation executions. The detailed algorithm used to generate the set of path conditions can be found in [14].

3.2 Verification Using the Symbolic Model Transformation Property Prover

Lúcio et al. [13] proposed a technique for the verification of *AtomicProperties* of DSLTrans transformations. We discuss how we redesigned the verification technique

proposed in [13] (Section 3.2.1) to facilitate verifying both *AtomicProperties* (Section3.2.2) and more complicated propositional formulae of *AtomicProperties* (Section 3.2.3).

3.2.1 Design of the Verification Technique

Figure 8 demonstrates the design of the verification technique used in our property prover. The abstract class *Property* represents the backbone of our verification technique and has an abstract function *verify*. Class *Property* encompasses two verification frameworks that can be used separately or together: class *PathConditionProperty* and class *BooleanProperty*.

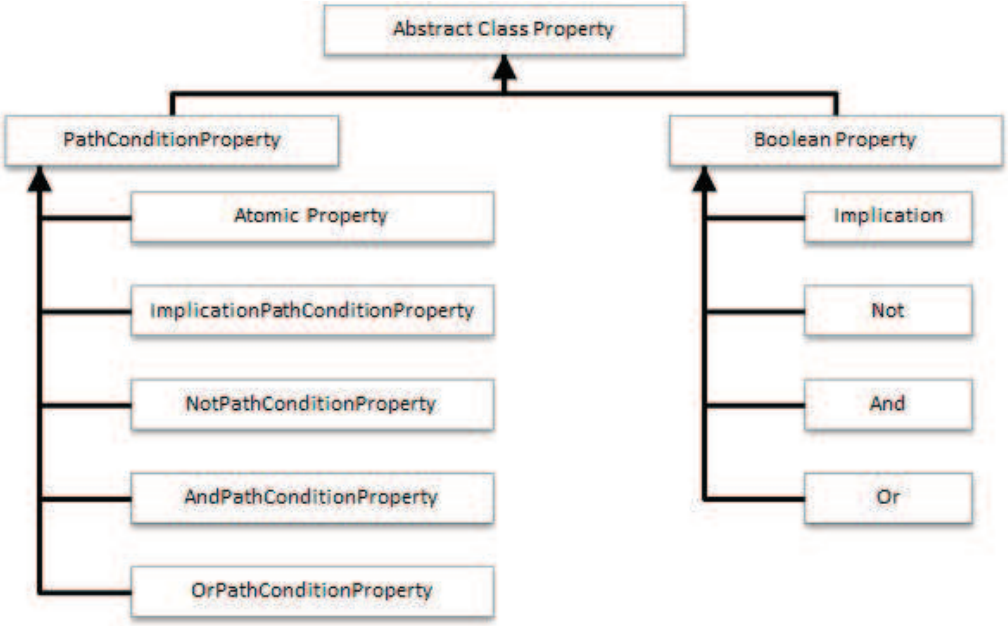

Figure 8: The design of the verification technique used in our symbolic model transformation property prover.

Class *PathConditionProperty*: Class *PathConditionProperty* is the framework used to verify a property (whether an *AtomicProperty* or a complex propositional logic formula of *AtomicProperties*) for each path condition in the generated set of path conditions (Section 3.1) and returns true if the property holds for all path conditions. Class *PathConditionProperty* has five subclasses: *AtomicProperty*, *ImplicationPathConditionProperty*, *NotPathConditionProperty*, *OrPathConditionProperty*, and *AndPathConditionProperty*.

Class *AtomicProperty* is used to verify that a simple, atomic property (explained in Section 2.2) holds for each path condition in the generated set of path conditions.

Class *ImplicationPathConditionProperty* is used to verify that an implication between two *PathConditionProperties* (i.e., any of its subclasses) holds for each path condition in the generated set of path conditions. Class *NotPathConditionProperty* is used to verify that a negation of a *PathConditionProperty* holds for each path condition in the generated set of path conditions. Class *OrPathConditionProperty* is used to verify that a disjunction between two *PathConditionProperties* holds for each path condition in the generated set of path conditions. Class *AndPathConditionProperty* is used to verify that a conjunction between two *PathConditionProperties* holds for each path condition in the generated set of path conditions.

Class *BooleanProperty*: Class *BooleanProperty* is used to evaluate propositional logic formulae with Boolean parameters, e.g., the parameters can be the results of verifying objects of class *PathConditionProperty* (explained above). Class *BooleanProperty* has four subclasses representing four Boolean propositional operators: *Implication*, *Not*, *Or*, and *And*. Each of the four subclasses is used to evaluate the corresponding propositional operator on its Boolean parameters.

3.2.2 Verifying *AtomicProperties*

The property prover's back-end component generates three preprocessed property files (as shown in Fig. 6) for every *AtomicProperty* (described in Section 2.2) to be verified:

1. Property file *Isolated* is a query transformation rule that matches the unconnected elements of the property's precondition.
2. Property file *Connected* is a query transformation rule that matches the property's precondition (i.e., the fully connected elements of the property's precondition).
3. Property file *Complete* is a query transformation rule that matches the complete property (i.e., the precondition and the postcondition of the property).

In what follows, we explain how the three preprocessed property files*(Isolated, Connected, Complete)* are used by the *verify* function of class *AtomicProperty* (explained in Section 3.2.1) to prove an *AtomicProperty* for the generated set of path conditions.

In general, for an *AtomicProperty* to hold for a transformation, the property should hold for all the generated path conditions of the transformation. In

other words, every path condition with a match pattern that satisfies the property's precondition must also have an apply pattern that satisfies the property's postcondition. Therefore, it is sufficient to find one path condition where the property does not hold (i.e., a path condition with a match pattern that satisfies the property's precondition and an apply pattern that does not satisfy the property's postcondition) to render the property false. This check is performed by the *verify* function of class *AtomicProperty*.

More specifically, the *verify* function of class *AtomicProperty* iterates through the generated set of path conditions (Section 3.1) and performs the following steps for every path condition i .

- *Step 1- Check if path condition i contains the isolated elements of the property's precondition as a subgraph:* This check is done by running the *Isolated* query transformation rule on path condition i. If the isolated elements of the property's precondition are not found in path condition i, then a new path condition is examined. If the isolated elements of the property's precondition are found in path condition i, then we *collapse* path condition i if it has rules with overlapping MatchModels and then we proceed to step 2. We demonstrate collapsing rules with overlapping MatchModels through an example. Assume that path condition i contains the two rules shown in Fig. 9. The MatchModels of the two rules can match two different Station objects in the input model or they can match the same Station object in the input model. We represent the latter case by collapsing the path condition containing the two rules in Fig. 9 into a new collapsed path condition containing the rule in Fig. 10[3]. The algorithm used to collapse rules is explained in detail in [14]. Collapsing is performed by the collapse transformation rules generated by the property prover's back-end component shown in Fig. 6.

- *Step 2- For each collapsed path condition resulting from path condition i, do the following:*

  - *(a) Check if the collapsed path condition contains the property's precondition as a subgraph:* This check is done by running the *Connected* query transformation rule on the collapsed path condition. If the property's precondition is not found in the collapsed path condition, then a new path condition is examined. If the property's precondition is found in the collapsed path condition , then we proceed to step 2b.
  - *b) Check if the collapsed path condition contains the complete property as a subgraph:* This check is done by running the *Complete* query

[3]In Fig. 9, we show the simplest case of collapsing where only two rules in a path condition have common MatchModels and can be collapsed into a new path condition. Our property prover handles more complicated cases where any number of rules in a path condition can have common MatchModels and the property prover generates all possible, collapsed path conditions.

transformation rule on the collapsed path condition. If the complete property is not found in the collapsed path condition, then the property does not hold for the transformation, the match pattern of the collapsed path condition is returned as a counter example, and no more path conditions are examined. If the complete property is found in the collapsed path condition, then the property holds for this path condition and a new path condition is examined.

Thus, after running the above steps, the property prover will render a property to be either true or false for the transformation of interest, with a counter example for the latter case.

Theoretically, collapsing must be done for each path condition after generating the set of path conditions. As an optimization step, we choose to collapse a path condition only after checking that the path condition has the isolated elements of the property's precondition because we found that collapsing is computationally expensive. Thus, if the isolated elements of the property's precondition do not exist in the path condition, then there is no need to collapse the path condition since the elements needed for the property do not exist and verifying the property for that path condition can be skipped.

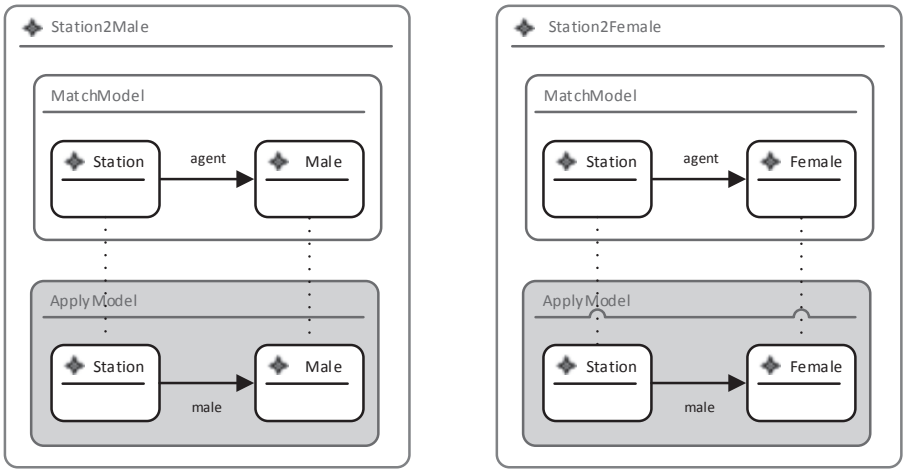

Figure 9: An example of two rules with common elements in their MatchModels that can be collapsed.

Two properties were demonstrated in Fig. 4 and Fig. 5 and explained in Section 2.2. Verifying the property in Fig. 4 using our property prover required the collapsed path condition (shown in Fig. 10) and returned true. Verifying the property in Fig. 5 using our property prover returned false when the property prover found one path condition whose match pattern has the property's precondition as a subgraph but does not have the complete property as a subgraph.

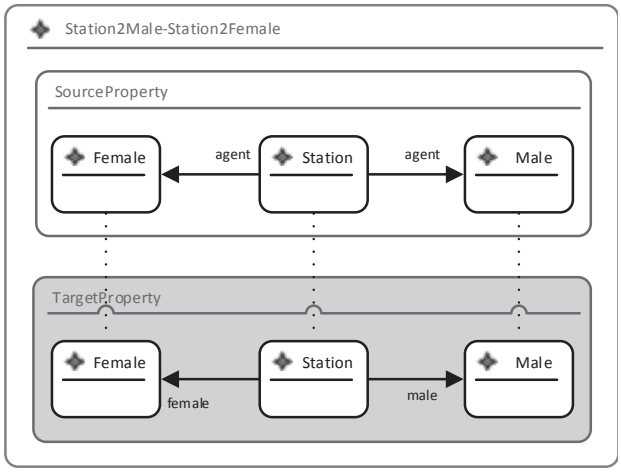

Figure 10: The result of collapsing the two rules in Fig. 9 into one rule.

3.2.3 Verifying Propositional Logic Formulae of Simple Properties

The new design of the symbolic model transformation property prover (explained in Section 3.2.1) facilitates the formulation and verification of properties in the form of propositional logic formulae of varying complexities. More specifically, calling the verify function on the subclasses of class *PathConditionProperty* (i.e., *ImplicationPathConditionProperty*, *NotPathConditionProperty*, *OrPathCondition-Property*, and *AndPathConditionProperty*) recursively calls the *verify* functions of their parameters (which are objects of type *PathConditionProperty*, including *AtomicProperties*). Similarly, calling the verify function on any of the subclasses of class *BooleanProperty* (i.e., *Implication*, *Not*, *Or*, and *And*) recursively calls the *verify* functions of their parameters (which are objects of type *BooleanProperty*). This enables composing complicated propositional logic formulae (whose parameters themselves could be other propositional logic formulae) and verifying them for the generated set of path conditions.

We demonstrate how the different subclasses of class *Property* (discussed in Section 3.2.1) can be used to formulate different propositional logic formulae and how do they differ in their interpretation in our property prover. Assume that N path conditions were generated for our transformation of interest (*PathCond1, PathCond2, ... PathCondN*) and that we have two *AtomicProperties* (namely, *AtomicProp1* and *AtomicProp2*). Further, assume a property expressed as the propositional logic formula in eqn. 1[4]:

$$AtomicProp1 \implies AtomicProp2 \tag{1}$$

[4]More complicated propositional logic formulae can be formulated. We stick to a simple example to keep the discussion of how different constructs are interpreted in our property prover a comprehensible one.

Using the subclasses of *PathConditionProperty*, the property shown in eqn.1 can be formulated as follows:

$$ImplicationPathConditionProperty(AtomicProp1, AtomicProp2) \quad (2)$$

The formulation in eqn. 2 is interpreted and verified by our property prover as follows:

$$\begin{aligned}
&\{(AtomicProp1 == true\ for\ pathCond1) \implies (AtomicProp2 == true\ for\ pathCond1)\}\ \&\\
&\{(AtomicProp1 == true\ for\ pathCond2) \implies (AtomicProp2 == true\ for\ pathCond2)\}\ \&\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots\\
&\{(AtomicProp1 == true\ for\ pathCondN) \implies (AtomicProp2 == true\ for\ pathCondN)\}
\end{aligned} \quad (3)$$

where each *AtomicProperty* is preprocessed by the property prover's back-end component into three property files (as shown in Fig. 6) and verified by the property prover's main component as described in Section 3.2.2. Thus, using the subclasses of *PathConditionProperty* implies that for the property to hold, eqn. 1 must be evaluated for each path condition separately and it must be true for all path conditions.

The property shown in eqn. 1 will have different semantics if we verify it using the subclasses of *BooleanProperty* by formulating the property as follows:

$$Implication(AtomicProp1, AtomicProp2) \quad (4)$$

The formulation in eqn. 4 is interpreted and verified by our property prover as follows:

$$\begin{aligned}
&\{AtomicProp1 == True\ \forall PathConditions(1 \ldots N)\} \implies\\
&\qquad \{AtomicProp2 == True\ \forall PathConditions(1 \ldots N)\}
\end{aligned} \quad (5)$$

where each *AtomicProperty* is preprocessed by the property prover's back-end component into three property files (as shown in Fig. 6) and verified by the property prover's main component as described in Section 3.2.2. Thus, using the subclasses of *BooleanProperty* implies that for the property to hold, *AtomicProp1* will be verified for $\{PathCond1\ \ldots PathCondN\}$. The same is done for *AtomicProp2*. Then, the implication operator will be applied to the results of verifying the two *AtomicProperties* separately, and the final result must be true.

In the previous examples, we demonstrated how we can use either the subclasses of *PathConditionProperty* or *BooleanProperty* to formulate and verify formulae that manipulate *AtomicProperties* as the basic units of the formula. We also showed how the semantics of both cases differ. Thus, it is up to the user to choose the subclasses to use for formulating and verifying properties based on the type of properties of interest.

While *PathConditionProperty* and *BooleanProperty* can be used separately for property verification, inheriting from class *Property* allows the user to use them together, too. For example, the *verify* function of an *Or* object (subclass of *BooleanProperty*) can take as parameters the results of verifying any two *PathConditionProperty* objects (e.g., a *NotPathConditionProperty* and an *AndPathConditionProperty*). Calling the verify function of the *Or* object will recursively call the verify function of the arguments based on their classes. For the case study discussed in Section 4, all properties were formulated and verified using only the subclasses of *PathConditionProperty*. Nevertheless, the design of the property prover gives the user the flexibility to formulate properties by mixing and matching subclasses of *PathConditionProperty* together with subclasses of *BooleanProperty*.

4 Industrial Case Study

Previously in [16], we reported on an industrial case study where we implemented a transformation that maps between subsets of the GM metamodel and the AUTOSAR metamodel as its source and target metamodels. In that work, we focused on subsets of the two metamodels that represent the deployment and interaction of software components. Later in [15], we proposed some properties of interest for our aforementioned GM-2-AUTOSAR model transformation.

To demonstrate the practicality of using our symbolic model transformation property prover, we use it to verify the properties proposed in [15] on the GM-2-AUTOSAR transformation [16] after reimplementing it in DSLTrans. In this section, we first overview the GM-2-AUTOSAR model transformation problem [16] and its reimplementation in DSLTrans (Section 4.1). Then, we summarize the properties of interest for the GM-2-AUTOSAR model transformation presented in [15] (Section 4.2). Finally, we discuss the formulation and verification of the aforementioned properties of interest using our property prover and the obtained results (Section 4.3).

4.1 GM-2-AUTOSAR Model Transformation

We overview the source GM metamodel and the target AUTOSAR metamodel of our GM-2-AUTOSAR transformation. We also discuss reimplementing the transformation in DSLTrans to achieve the required mapping between the two metamodels.

The GM Metamodel: Fig. 11 illustrates the subset of the GM metamodel that we manipulated in our transformation in [16][5]. The *PhysicalNode* models a physical node on which software is deployed. A *PhysicalNode* may contain multiple *Partition*s (i.e., processing units or memory partitions) on which software is deployed. 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 and/or requiring *Service*s to/from these behavior-encapsulating entities.

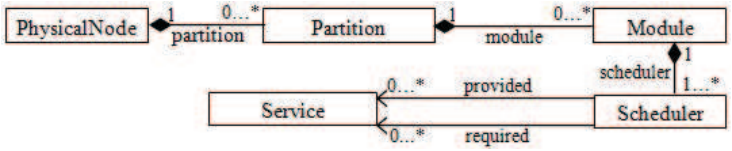

Figure 11: Subset of the GM metamodel directly used by our transformation in [16].

The AUTOSAR Metamodel: The *System* template [2] is one of the templates in the AUTOSAR metamodel that is used to model the configuration of a system or an Electronic Component Unit (ECU). An ECU is a physical unit on which software is deployed. When used for modeling an ECU's configuration, the System template is referred to as the *ECU Extract*. Fig. 12 shows the subset of the ECU Extract manipulated by our transformation.

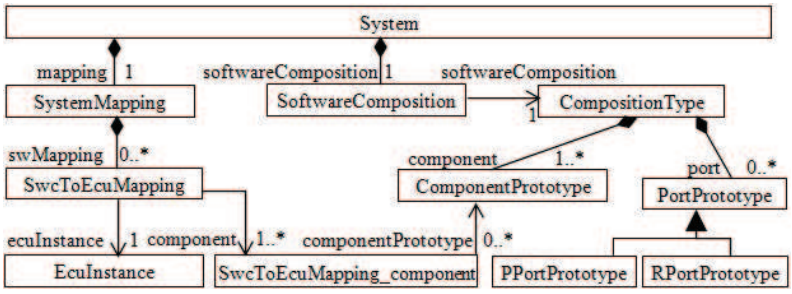

Figure 12: Subset of the AUTOSAR System Template directly used by our transformation.

The ECU extract is modeled using the *System* element that aggregates *SoftwareComposition* and *SystemMapping* elements. *SoftwareComposition* points to *CompositionType* which eliminates any nested software components in a *SoftwareComposition*. *SoftwareComposition* models the architecture of the software components deployed on an ECU, their ports, and the ports' connectors. Each

[5]We follow the same obfuscated naming conventions that we used for the GM metamodel in [16] for reasons of confidentiality.

software component is modeled using a *ComponentPrototype*; each port is modeled using the *PortPrototype* type (i.e., a *PPortPrototype* or a *RPortPrototype* for providing or requiring data and services).

SystemMapping binds the software components to ECUs (using *SwcToEcuMapping* elements) and data elements to signals and frames (not shown). *SwcToEcuMapping* elements assign *SwcToEcuMapping_component*s to an *EcuInstance*. *SwcToEcuMapping_component*s, in turn, refer to *ComponentPrototype* elements. According to AUTOSAR, only one *SwcToEcuMapping* should be created for each processing unit or memory partition in an ECU.

Reimplementation of GM-2-AUTOSAR Transformation in DSLTrans: Since our symbolic model transformation property prover can only verify DSLTrans transformations, we reimplemented the GM-2-AUTOSAR transformation in DSLTrans. Table 1 shows the rules in each layer of the GM-2-AUTOSAR DSLTrans transformation, the input types mapped by each rule, and the generated output types. Rules of the first and third layers create output model elements. Rules of the second layer generate associations between elements previously created by the the first layer (shown in the actual transformation using backward links). Thus, the input types and output types shown for the rules of the second layer are actually types that have already been matched and created and for which the rules of the second layer create associations. The implemented transformation rules achieve the required mapping between the two metamodels that we described in detail in [16].

Layer	*Rule Name*	*Input Types*	*Output Types*
1	MapPhysNode2FiveElements	PhysicalNode	System, SystemMapping, SoftwareComposition, CompositionType, EcuInstance
	MapPartition	Partition	SwcToEcuMapping
	MapModule	Module	SwCompToEcuMapping_component, ComponentPrototype
2	ConnPhysNode2Partition	PhysicalNode, Partition	SystemMapping, EcuInstance, SwcToEcuMapping
	ConnPartition2Module	PhysicalNode, Partition, Module	CompositionType, ComponentPrototype, SwcToEcuMapping, SwCompToEcuMapping_component
3	ConnectPPortPrototype	Scheduler	PPortPrototype
	ConnectRPortPrototype	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.

4.2 GM-2-AUTOSAR Model Transformation Properties

In [15], we stated that properties of interest could be either *invariants* or *contracts*. Invariants are properties defined on the target metamodel elements only. While

contracts are properties that relate source and target metamodel elements. Based on these definitions, we further defined four categories of properties in [15] that we are interested in verifying for the GM-2-AUTOSAR model transformation: *Multiplicity Invariants*, *Uniqueness Contracts*, *Security Invariants*, and *Pattern Contracts*. For each category, we formulated several properties that are summarized in Table 2 as OCL properties. We add an abbreviation to the beginning of each property in Table 2 (e.g., (M1), (S1)) that will be used to refer to the property in the rest of this paper. We omit Uniqueness Contracts for this study since they are contracts that reason about attribute values. This requires handling data types during the generation of path conditions which we have not yet implemented.

The six multiplicity invariants ensure that the transformation's output does not violate the multiplicities in the AUTOSAR metamodel (Fig. 12). The only security invariant defined, S1, mandates that within any *System* element, all its composite *SwcToEcuMapping*s must refer to *ComponentPrototype*s that are contained within the *CompositionType* lying under the same *System* element (refer to Fig. 12). The two pattern contracts require that if a certain pattern of elements is found in the input model, then a corresponding pattern of elements must be found in the output model. A detailed explanation of each property in Table 2 can be found in [15].

4.3 Verifying Properties of the GM-2-AUTOSAR Model Transformation

We used our symbolic model transformation property prover to verify the 9 properties in Table 2 for the DSLTrans version of the GM-2-AUTOSAR transformation (Section 4.1). The two pattern contracts were expressed as *AtomicProperties* where the precondition encompassed the pattern of input model elements and the postcondition encompassed the pattern of output model elements. The six Multiplicity Invariants and the Security Invariant were expressed using a combination of *ImplicationPathConditionProperty*, *AndPathConditionProperty*, and *NotPathConditionProperty* that manipulate *AtomicProperties* as their basic units.

We demonstrate the formulation of property *M1* from Table 2 using our property prover. Property *M1* ensures that each *CompositionType* is associated to one or more *ComponentPrototype*s through the *component* association. Property *M1* can be restated as follows "if a *CompositionType* is created by the GM-2-AUTOSAR transformation, then this *CompositionType* must be connected to at least one *ComponentPrototype*". Fig. 13 shows the general idea of how such a property can be expressed using two *AtomicProperties*.

There are several points to note about Fig. 13. First, the two *AtomicProperties* used have empty preconditions. DSLTrans gives us the flexibility to express invariants on the transformation's output by allowing empty preconditions, i.e.,

Multiplicity Invariants:

- ($M1$) **Context** CompositionType **inv** CompositionType_component: self.component→**size()** ≥ 1
- ($M2$) **Context** SoftwareComposition **inv** SoftwareComposition_softwareComposition: self.softwareComposition $\neq$ **null**
- ($M3$) **Context** SwcToEcuMapping **inv** SwcToEcuMapping_component: self.component→**size()** ≥ 1
- ($M4$) **Context** SwcToEcuMapping **inv** SwcToEcuMapping_ecuInstance: self.ecuInstance $\neq$ **null**
- ($M5$) **Context** System **inv** System_softwareComposition: self.softwareComposition $\neq$ **null**
- ($M6$) **Context** System **inv** System_mapping: self.mapping $\neq$ **null**

Security Invariant:

- ($S1$) **Context** System **inv** Self_Cont: mapping.swMapping→**forAll**(swc2ecumap: SwcToEcuMapping| swc2ecumap.component → **forAll**(mapcomp : SwCompToEcuMapping_component| mapcomp.componentPrototype→**forAll**(comppro: ComponentPrototype| softwareComposition.softwareComposition.component→ **exists**(c: ComponentPrototype| c=comppro))))

Pattern Contracts:

- ($P1$) **Context** Global **inv** Sig2P: PhysicalNode.**allInstances()**→ **forAll**(e1:PhysicalNode| e1.partition→ **forAll**(vd: Partition| vd.module→ **forAll**(di: Module| di.scheduler→ **forAll**(ef:Scheduler| (ef.provided→**notEmpty()**) **implies** (System.**allInstances()**→**one**(sy:System| (sy.shortName=e1.Name) **and** (sy.softwareComposition.softwareComposition.port→ **one**(pp:PortPrototype| (pp.shortName=ef.Name) **and** (pp.**oclIsTypeOf**(PPortPrototype)))))))))
- ($P2$) **Context** Global **inv** Sig2R: PhysicalNode.**allInstances()**→ **forAll**(e1:PhysicalNode| e1.partition→ **forAll**(vd:Partition| vd.module→ **forAll**(di: Module| di.scheduler→ **forAll**(ef:Scheduler| (ef.required→**notEmpty()**) **implies** (System.**allInstances()**→ **one**(sy:System| (sy.shortName=e1.Name) **and** (sy.softwareComposition.softwareComposition.port→ **one**(rp:PortPrototype| (rp.shortName=ef.Name) **and** (rp.**oclIsTypeOf**(RPortPrototype)))))))))

Table 2: Formulated OCL Constraints

empty preconditions match on any input model elements. Second, we used two new attributes called *pivotOut* and *pivotIn* for the *CompositionType* class in the postcondition of the two *AtomicProperties*. Attribute *pivotOut* is used to give the *CompositionType* element matched by *AtomicProperty1* a marker (or a pivot) such that we can refer to that specific element in *AtomicProperty2* by giving both *pivotIn* and *pivotOut* the same value (e.g., 'elem1' as shown in Fig. 13). Finally, we can not use *AtomicProperty2* only to express property *M1* since this implies that we mandate that every output has a *CompositionType* connected to at least one *ComponentPrototype*. Expressing *M1* using the implication shown in Fig. 13 will give us the correct interpretation since it mandates that only if the output has a *CompositionType*, then that specific *CompositionType* (as per the *pivotIn* and *pivotOut* attribute values) must be connected to at least one *ComponentPrototype*.

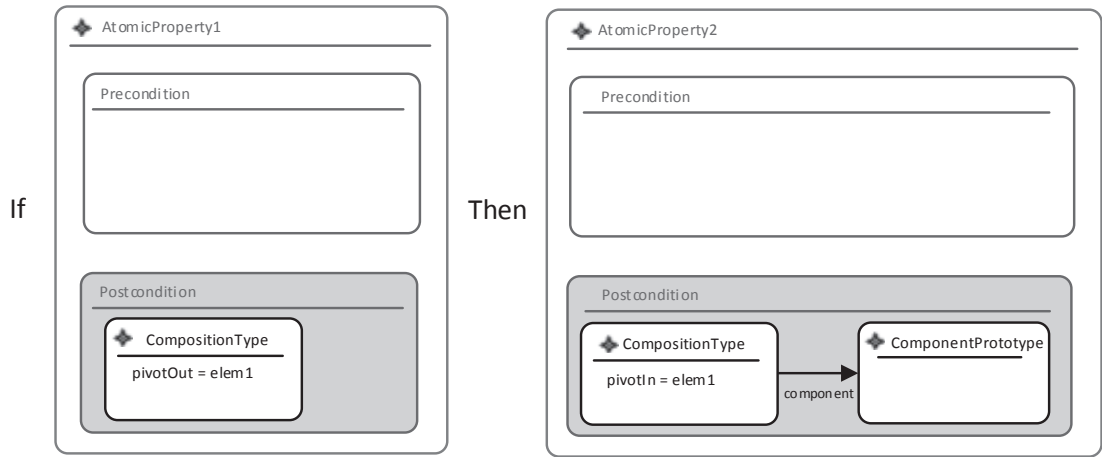

Figure 13: Expressing property *M1* using two *AtomicProperties*.

To express property *M1* using our property prover, we first built the two *AtomicProperties* shown in Fig. 13. Then, from the classes shown in Fig.8, we formulated the implication shown in Fig. 13 using *ImplicationPathConditionProperty* as follows:
$\texttt{I}mplicationPathConditionProperty(AtomicProperty1, AtomicProperty2).verify$
The rest of the properties shown in Table 2 were formulated in a similar manner and were verified as discussed in Section 3.2. Verification of all 9 properties returned 'true'. This implies that our DSLTrans version of the GM-2-AUTOSAR model transformation will always generate output models that do not violate the six multiplicity invariants, the security invariant, and the two pattern contracts shown in Table 2[6].

To assess the performance of our property prover, we measured the time taken to generate the set of path conditions and to verify each of the 9 properties shown in Table 2. The property prover took 0.8223 seconds to generate the set of path conditions for the GM-2-AUTOSAR transformation. Table 3 shows the time taken by our property prover (in seconds) to verify each of the 9 properties shown in Table 2 using the generated set of path conditions. The longest time taken to verify a property was 0.0243 seconds (for property *P1*). Thus, using our symbolic model transformation property prover to verify properties for an industrial transformation was successful and execution times were very short (as shown in Table 3). More experiments are required before we can confidently claim that verification using our property prover scales well to transformations of varying complexities.

[6]We verified the properties on the non-faulty version of the model transformation, i.e., after fixing the two bugs in the transformation found in [15]. Our next step, would be to check the properties on the faulty version of the GM-2-AUTOSAR transformation too to ensure that our property prover can uncover the same bugs that were uncovered by the prototype that we used in [15].

Property	M1	M2	M3	M4	M5	M6	S1	P1	P2
Verification Time (Seconds)	0.0151	0.0213	0.0218	0.0217	0.0222	0.0230	0.0201	0.0243	0.0240

Table 3: The time taken by our symbolic model transformation property prover (in seconds) to verify each of the 9 properties shown in Table 2.

5 Enabling Technology

When deciding on a model transformation framework to build our symbolic model transformation property prover, three main aspects were taken into consideration. First, as discussed in the previous sections, a lot of the work done by our property prover to verify DSLTrans transformation properties relies on graph matching and graph rewriting. Second, support for higher order transformations (HOTs) is necessary to facilitate developing the back-end component of the property prover (summarized in Appendix A and described in detail in [14]). Thus, a transformation language with an explicit metamodel is required. Finally, a language with a control flow mechanism is needed to allow scheduling model transformation rules.

After considering the above mentioned points, we have chosen to implement our symbolic model transformation property prover using a mix of Python and the T-Core framework [19].

T-Core is a set of primitive model transformation blocks that can be used to replicate the behavior of existing model transformation languages (e.g. to compare their expressiveness and to provide a framework for interoperability) or to build new model transformation languages. The framework includes five primitive transformation blocks that exchange models and transformation-related information in messages called *packets*. These five primitive transformation blocks are the *Matcher*, the *Iterator*, the *Rewriter*, the *Rollbacker*, and the *Resolver*.

The Matcher finds matches of a given precondition pattern within a model by running an efficient combination of the Ullmann's [21] and VF2's [10] subgraph isomorphism algorithm and stores those matches in a packet. The Iterator allows selecting the next matched submodel from the set of matches stored in a packet such that the submodel to be changed can be locked on. The Rewriter consumes a matched subgraph from a model in a packet and changes the model according to a given postcondition pattern. The Rollbacker allows backtracking by checkpointing and restoring packets. The Resolver facilitates solving potential conflicts between matches and rewritings. An additional, sixth construct called the *composer* is used to encapsulate compositions of the five, aforementioned primitive transformation blocks. The goal of the composer is to create complex transformation blocks (e.g., querying or rewriting all the matches found for a precondition) from the five primitive transformation blocks.

Since T-Core is a Python library, we interleaved T-Core primitives with Python

code to develop our property prover. The backward link matchers, the collapse transformation rules, and the preprocessed property files generated by the property prover's back-end component (shown in Fig. 6) generate the necessary patterns that are used to initialize T-Core primitives. The initialized T-Core primitives manipulate the preprocessed DSLTrans model transformation rules (generated by the property prover's back-end component as shown in Fig. 6) and are scheduled using Python code (according to the steps described in previous sections) to generate the set of path conditions and verify the property of interest.

6 Related Work

While numerous studies investigated different model transformation verification techniques [17, 4], we only review studies that proposed verification techniques similar to the one used by our property prover where all the possible transformation executions are generated and verified with respect to a property. We also review the few studies that investigated industrial model transformation validation and verification.

Büttner et al. [8] and Cabot et al. [9] proposed translating the transformation of interest and its manipulated metamodels into a transformation model and used model finders (e.g., UML2Alloy) and constraint solvers (e.g., UMLtoCSP) to verify the transformation with respect to a specific property. Anastasakis et al. [5] and Baresi and Spoltini [6] translated the transformation of interest and its manipulated metamodels into an Alloy model and used the Alloy Analyser to verify the translated transformation within a specific scope. Troya and Vallecillo [20] translated the transformation of interest into Maude and used the analysis capabilities of Maude to verify the possible executions of the input transformation.

The applicability of validation and verification tools to industrial model transformations has been rarely investigated in the literature albeit its importance. In [15], we used the prototype tool proposed in [8] to check the satisfiability of a relational representation (or a transformation model) of our GM-to-AUTOSAR transformation with respect to the same OCL properties that we presented in Section 4.2. The tool was found to scale well for scopes as big as 12. In another study [18], we validated our GM-2-AUTOSAR transformation using a black-box testing tool proposed by Fleurey et al. [11, 12]. We found that many of the generated test cases did not trigger the transformation's rules. Wang et al. [22] proposed using an optimization algorithm to generate test cases for model transformation testing and demonstrated their approach on a case study from the banking sector.

Difference between our study and the surveyed studies: Our study differs from the surveyed studies in several aspects. First, verification is performed on

an intuitive graph-based transformation language that does not require a rigorous mathematical background to be used, unlike Maude. Second, we demonstrated the practicality of using our property prover by using it to verify both a simple model transformation and an industrial model transformation.

In another ongoing study, we have also successfully proved two main results for our property prover, which are rarely proved in similar studies:

- *Validity and completeness* of the generated set of path conditions.
- *Validity and completeness* of our property prover's verification technique.

Validity of the generated set of path conditions means that every path condition represents at least one transformation execution. Completeness of the generated set of path conditions means that every transformation execution is represented by at least one path condition. Many of the surveyed studies translated their transformation into another formalization and verified the properties of interest on the transformation in the latter formalization [8, 9, 5, 6, 20]. In such approaches, the translation done to the model transformation into another formalization needs to be itself verified before verifying the properties of interest, i.e., the validity of their approaches still need to be proved.

Validity of the verification technique means that verifying the property of interest for a path condition is equivalent to verifying the property for a transformation execution that corresponds to the aforementioned path condition. Completeness of the verification technique means that the property of interest is verified for all possible transformation executions. This is in contrast with the study we conducted in [15] where the performed verification is restricted to a certain scope and with model transformation testing which does not guarantee that the transformation is fault-free (i.e., testing only guarantees that the transformation runs correctly for the input test cases).

7 Conclusion and Future Work

In this study, we proposed a symbolic model transformation property prover that can be used to prove both *AtomicProperties* and more complicated propositional logic formulae of *AtomicProperties* for DSLTrans transformations. The property prover basically generates the set of path conditions (i.e., set of possible input patterns and their corresponding output patterns) for the transformation of interest and iterates over the set of path conditions to verify properties of varying complexities. If a property holds for all path conditions, then it will always hold for the transformation. Otherwise, the property does not hold for the

transformation and a counter example is generated. We demonstrated the property prover on the Police Station transformation (Fig. 2) as a running example. Then, we experimented with our property prover on an industrial case study that we previously presented in [16] and reported on the results obtained.

Several points can be addressed as future work. First, more experiments need to be performed to test the scalability of the property prover by using it to verify more complicated model transformations. Second, we currently provide the property to be verified and the required parameters to the property prover within the code of our prototype. Thus, the prototype requires an interface that accepts the property to be verified and the required parameters in some user-friendly graphical syntax that can then be interpreted and used within the property prover's code.

References

[1] DSLTrans User Manual. http://msdl.cs.mcgill.ca/people/levi/files/DSLTransManual.pdf.

[2] AUTOSAR Consortium. AUTOSAR System Template, `http://{AUTOSAR}.org/index.php?p=3&up=1&uup=3&uuup=3&uuuup=0&uuuuup=0/{AUTOSAR}_{TPS}_{S}ystem{T}emplate.pdf`, 2007.

[3] AUTOSAR Consortium. AUTOSAR, `http://{AUTOSAR}.org/`, 2007.

[4] Moussa Amrani, Levi Lúcio, Gehan Selim, Benoit Combemale, Jüergen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *Proceedings of the First Workshop on Verification And Validation of Model Transformations (VOLT)*, pages 921–928. IEEE Computer Society, 2012.

[5] K. Anastasakis, B. Bordbar, and J.M. Küster. Analysis of Model Transformations via Alloy. *MoDeVVa*, pages 47–56, 2007.

[6] Luciano Baresi and Paola Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *ICGT*, volume 4178 of *LNCS*, pages 306–320, 2006.

[7] Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *Software Language Engineering (SLE)*, pages 296–305. Springer, 2011.

[8] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL Transformations Using Transformation Models and Model Finders. In *ICFEM*, volume 7635 of *LNCS*, pages 198–213, 2012.

[9] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and Validation of Declarative Model-to-Model Transformations Through Invariants. *Systems and Software*, 83(2):283–302, 2010.

[10] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, October 2004.

[11] F. Fleurey, B. Baudry, P.A. Muller, and Y.L. Traon. Qualifying Input Test Data for Model Transformations. In *Software and Systems Modeling (SoSym)*, volume 8, pages 185–203, 2009.

[12] IRISA. Metamodel Coverage Checker (MMCC). http://www.irisa.fr/triskell/Software/protos/MMCC, 2012.

[13] Levi Lúcio, Bruno Barroca, and Vasco Amaral. A Technique for Automatic Validation of Model Transformations. In *Model Driven Engineering Languages and Systems*, pages 136–150. Springer, 2010.

[14] Levi Lúcio and Hans Vangheluwe. Symbolic Execution for the Verification of Model Transformations. Technical report, Technical Report SOCS-TR-2013.2, McGill U., 2013. http://msdl. cs. mcgill. ca/people/levi/files/MTSymbExec. pdf.

[15] Gehan Selim, Fabian Büttner, James R Cordy, Juergen Dingel, and Shige Wang. Automated Verification of Model Transformations in the Automotive Industry. In *16th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2013.

[16] Gehan Selim, Shige Wang, James R Cordy, and Juergen Dingel. Model Transformations for Migrating Legacy Models: An Industrial Case Study. *ECMFA*, pages 90–101, 2012.

[17] Gehan M.K. Selim, James R. Cordy, and Juergen Dingel. Analysis of Model Transformations. Technical Report 2012-592, Queen's University, 2012.

[18] Gehan MK Selim, Shige Wang, James R Cordy, and Juergen Dingel. Model Transformations for Migrating Legacy Deployment Models in the Automotive Industry. *Software and Systems Modeling*, pages 1–17, 2013.

[19] Eugene Syriani and Hans Vangheluwe. De-/re-constructing Model Transformation Languages. *Electronic Communications of the EASST*, 29, 2010.

[20] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10:5: 1–29, 2011.

[21] Julian R Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, January 1976.

[22] Wenquan Wang, Marouane Kessentini, and Wei Jiang. Test Cases Generation for Model Transformations from Structural Information. In *17th European Conference on Software Maintenance and Reengineering*, Genova, Italy, 5-8 March 2013.

A

The Back-End Component of the Symbolic Model Transformation Property Prover

As explained in Section 3, our symbolic model transformation property prover takes four inputs: the DSLTrans model transformation of interest, the source and target metamodels, and the property to prove. These four inputs are first preprocessed by a preprocessing component of the property prover (Fig. 6) so that they can be used for the generation of the set of path conditions and for the property verification. In other words, the preprocessing step *customizes* the property proverfor the verification of a specific transformation.

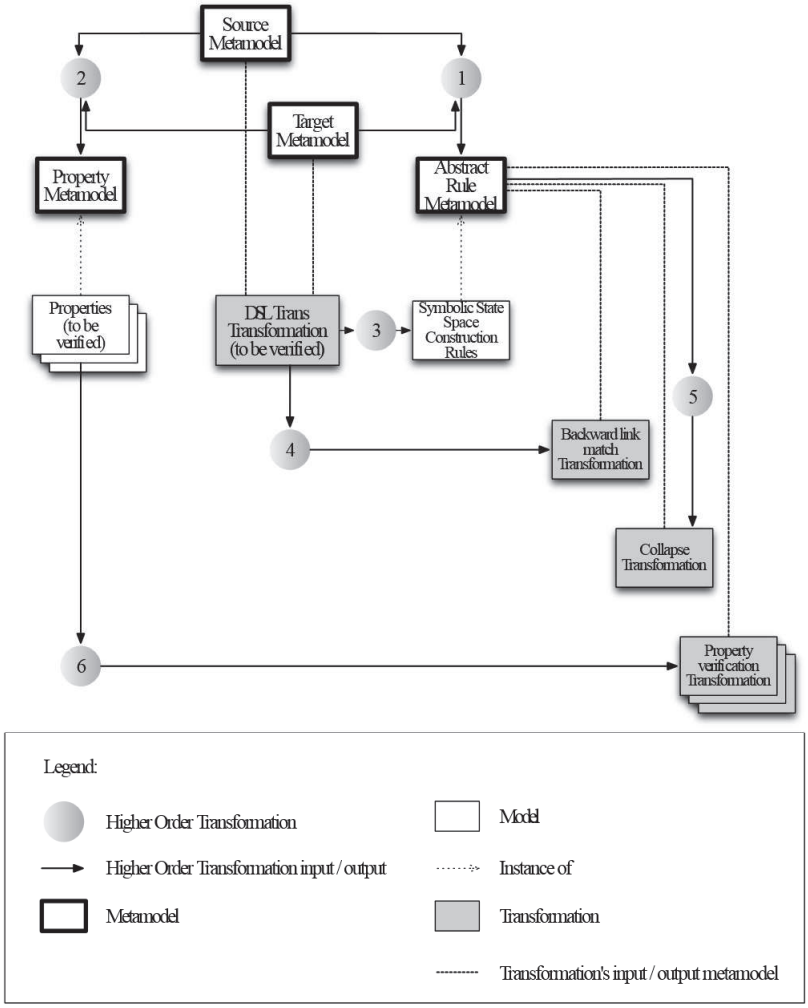

Figure 14: The six higher order transformations (HOTs) executed by the preprocessing component of our symbolic model transformation property prover.

The preprocessing step of our property prover is composed of six higher order transformations (HOTs). Fig. 14 demonstrates these six HOTs and their manipulated artifacts[7].

1. **Generate the abstract rule metamodel:** This HOT takes as input the source and target metamodels of the transformation of interest and returns a metamodel which is used to generate an abstract form of the transformation's rules. Such a metamodel for the Police Station transformation can be observed in Fig. 15. These abstract rules will be later used as the basic building blocks for generating the set of path conditions.

2. **Generate the property metamodel:** This HOT takes as input the source and target metamodels of the transformation of interest and builds the metamodel in which properties are expressed.

3. **Generate the symbolic state space construction rules:** this HOT builds a set of models corresponding to the abstract form of the rules of the transformation of interest. These abstract rules will be used to generate the set of path conditions. The rules generated by this HOT are instances of the abstract rule metamodel. The rules used in all path conditions are symbolic state space construction rules.

4. **Generate the backward link match transformation:** This HOT builds the query transformation that checks whether a graph including backward links exists in an abstract rule. The input metamodel of the backward link match transformation is the abstract rule metamodel.

5. **Generate the collapse transformation:** This HOT takes as input the abstract rule metamodel generated in the first HOT and generates the collapse rules for abstract rules which are instances of the abstract rule metamodel. The collapse transformation has as both source and target metamodel the abstract rule metamodel.

6. **Generate the property verification transformation:** this HOT generates three query transformations for each property to be verified: (i) a query transformation that checks whether the unconnected precondition elements of the property are present in a symbolic state; (ii)a query transformation that checks whether the whole precondition of the property is a subgraph of the match pattern of a possibly collapsed path condition; (iii) a query transformation that checks whether the whole property is a subgraph of a possibly collapsed path condition.

A more detailed explanation of the six HOTs can be found in [14].

[7]The six HOTs are currently being developed. To continue with our experiments, we manually performed the tasks done by the six HOTs for our industrial case study discussed in Section 4.

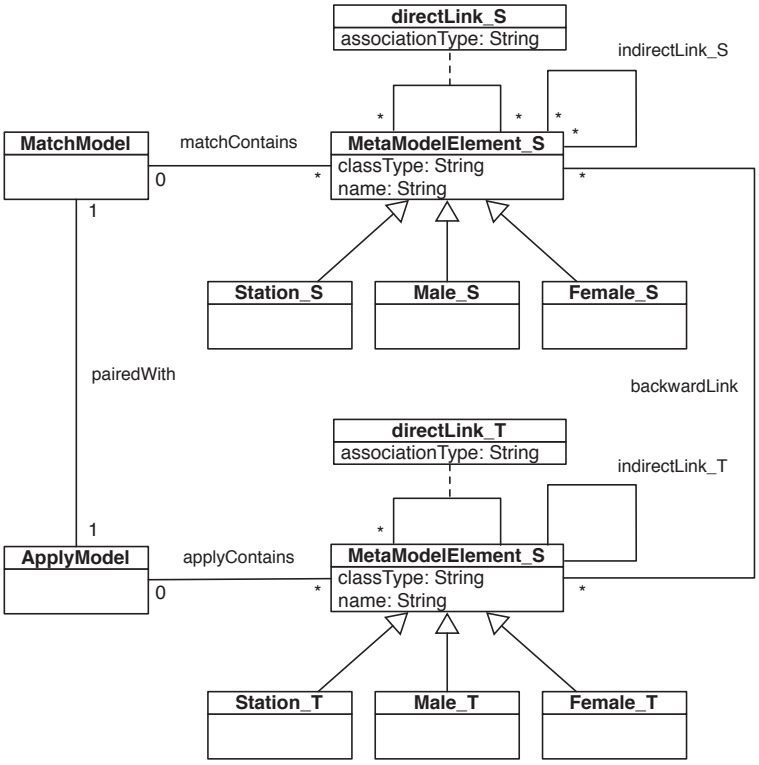

Figure 15: The abstract rule metamodel for the Police Station transformation.