Symbolic Execution for the Verification of Model Transformations

Levi Lúcio[†] and Hans Vangheluwe[†‡]

[†]School of Computer Science, McGill University, Canada
[‡]University of Antwerp, Belgium
{levi,hv}@cs.mcgill.ca

Abstract. In this paper we present a novel technique and a prototype implementation for proving properties of model transformations expressed in the DSLTrans language. The approach is based on symbolic execution and the properties we are interested in concern relations between the structure of the input and output models. In particular, the properties are implications of the form 'if a structural relation between some elements of the source model holds, then another structural relation between some elements of the target model must also hold'. Our technique is transformation dependent but model independent, meaning the proofs we produce hold for all executions of a given DSLTrans transformation specification running on any instance of the transformation's input metamodel.
Our proof technique is based on (i) building the finite symbolic execution for a given DSLTrans transformation and (ii) on checking whether the property holds for all elements of the finite symbolic execution execution set. We explain how a symbolic execution and a proof is built with our technique by using model transformations written using our T-Core framework. Furthermore we apply our tool to the proof of properties of an example transformation and present some performance results for our approach.

1 INTRODUCTION

In 2003 Sendall and Kozaczynski proposed model transformations as the *heart and soul* of model driven software development [1]. Since then many model transformation languages have emerged and are being used intensively not only in MDD, but also in other scopes, for example when software development processes benefit from formalised translators between languages. Due to their practicality and appropriate level of abstraction, model transformations are at the moment of the writing of this paper the stardard means for performing computations on models.

Authors such as Mens, Czarnecki and Van Gorp have called for the development of verification, validation and testing techniques for model transformations in their well know paper 'A Taxonomy of Model Transformations' [2] from 2006. Despite the many publications on this topic since then, the field of analysis of model transformations seems to be still in its (late) infancy, as evidenced by [3].

The research presented in this paper follows from our proposals in [4] and [5]. In [4] we have formally introduced the DSLTrans transformation language. DSLTrans

is Turing Incomplete, as it avoids constructs which imply unbounded recursion or non-determinism. Despite this *expressiveness reduction*, we have shown via several examples [6–8] that DSLTrans is sufficiently expressive to tackle typical translation problems. Our verification transformation technique is based on the theory introduced in [5], where we have described how to abstractly build a symbolic execution for DSLTrans transformations. Additionally, in [5], we have mathematically proved that such a symbolic execution is finite, given an abstraction over the number of times the transformation's rules match on concrete elements of input models. This finiteness a necessary condition for our technique to be applicable.

The properties we aim at proving are *model syntax relations* [3]. Such properties are essentially precondition-postcondition axioms involving statements about the input and output models of a transformation. Several authors have explored this kind of transformation properties [9–12]. According to the classification presented in [3], our technique is *transformation dependent* and *input independent*, meaning we can prove properties hold for all executions of a given model transformation.

In this paper we present concrete algorithms for the technique originally presented in [5]. As for any exhaustive proof technique (making use of an abstraction to render the search space finite), the main problem to deal with is state space explosion. Additionally, graph matching and rewriting is heavily used to build our symbolic states. Given the subgraph isomorphim problem is NP-Complete, that additional level of algorithmic complexity is added our problem. In order to tackle this issue we have used model transformations for the construction of the symbolic execution for a given model transformation as well as for the proof algorithms. Model transformations allow us to seamlessly and efficiently deal with all the parts of the technique that require graph manipulation. We will provide performance results obtained from our prototype implementation and analyse the scalability our approach in order to infer its applicability to real world problems.

This paper is organised as follows: section 2 briefly introduces the DSLTrans model transformation language and the running example we will use throughout the paper; in section 3 we introduce our property language and the algorithms for symbolic execution construction and property proof; section 4 enumerates all the necessary artifacts such that the algorithms presented in section 3 can be ran on any DSLTrans transformation; section 5 introduces the T-Core model transformation framework which allows us to cope with the necessary graph manipulations; in section 6 we discuss implementation details and present scalability results; section 7 discusses the results and presents future work; section 8 presents the related work and finally in section 9 we conclude.

2 The DSLTrans Transformation Language

In what follows we will present the DSLTrans running example we will use throughout this paper. As we present the running example we will also introduce the DSLTrans language itself and its constructs.

Figure 1 presents two metamodels of languages for describing views over the organization of a police station. The metamodel annotated with 'Organization Language' represents a language for describing the chain of command in a police station, which in-

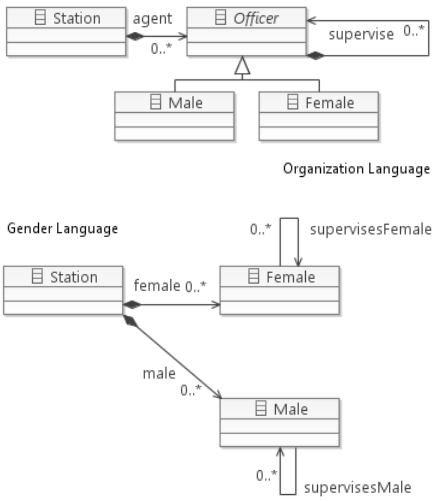

Fig. 1: Metamodels for describing police stations' chains of command (top) and police stations' gender classification views (bottom)

cludes male (*Male* class) and female officers (*Female* class). The metamodel annotated with 'Gender Language' represents a language for describing a different view over the chain of command, where the officers working at the police station are classified by gender. In figure 2 we present a transformation written in DSLTrans[1] between models of both languages. The purpose of this transformation is to flatten a chain of command given in language 'Organization Language' into two independent sets of male and female officers. Within each of those sets the command relations are kept, i.e. a female officer will be directly related to all her female subordinates and likewise for male officers. In the text that follows we will call this transformation the *Police Station* transformation.

An example of an instance of this transformation can be observed in figure 3, where the original model is on the left and the transformed one on the right. Notice that the elements s, m_k and f_k in the figure on the left are instances of the source metamodel elements $Station$, $Male$ and $Female$ respectively (in figure 1). The primed elements in the figure on the right are their instance counterparts in the target metamodel.

A transformation in DSLTrans is formed by a set of input model sources called *file-ports* (e.g. 'inputSquad.xmi' in figure 2) and a list of *layers* (e.g. 'Entities' and 'Relations' layers in figure 2). Both layers and file-ports are typed according to metamodels. DSLTrans executes sequentially the list of layers of a transformation specification. A layer is a set of transformation rules which executes in a non-deterministic order but produce a deterministics result. Each transformation rule is a pair $(match, apply)$

[1] DSLTrans has been implemented as an Eclipse plug-in [7]. The example shown in figure 2 is expressed using DSLTrans' concrete visual syntax in the Eclipse editor.

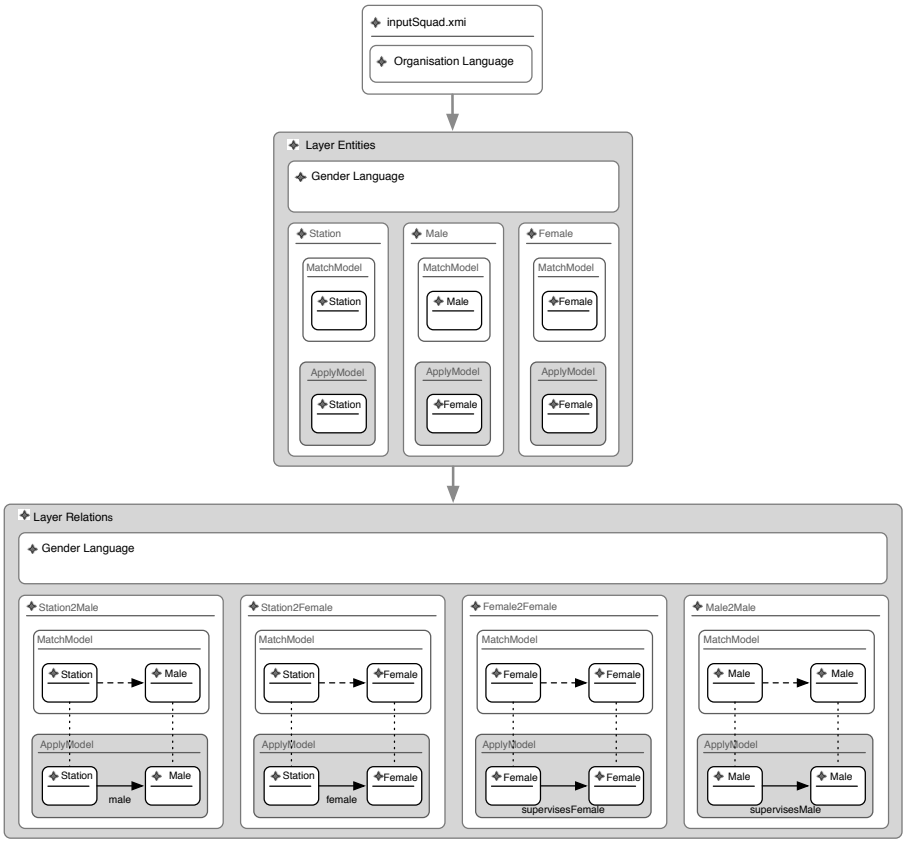

Fig. 2: A model transformation expressed in DSLTrans

where *match* is a pattern holding elements from the source metamodel, and *apply* is a pattern holding elements of the target metamodel. For example, in the transformation rule 'Stations' in the 'Entities' layer (in figure 2) the *match* pattern holds one 'Station' class from the 'Squad Organization Language' metamodel — the source metamodel; the *apply* pattern holds one 'Station' class from the 'Squad Gender Language' metamodel — the target metamodel. This means that all elements in the input source which are of type 'Station' of the source metamodel will be transformed into elements of type 'Station' of the target metamodel.

Let us first define the constructs available for building transformation rules' match patterns. We will illustrate the constructs by referring to the transformation in figure 2.

– *Match Elements*: are variables typed by elements of the source metamodel which can assume as values elements of that type (or subtype) in the input model. In our example, a match element is the 'Station' element in the 'Stations' transformation rule of layer 'Entities' layer;

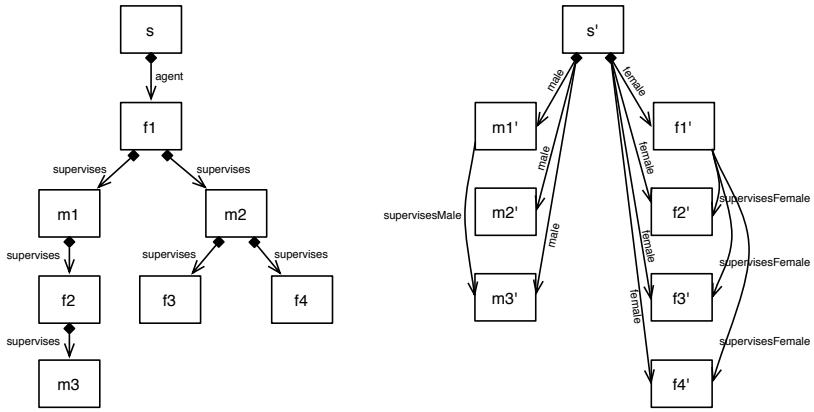

Fig. 3: Original model (left) and transformed model (right)

- *Attribute Conditions*: conditions over the attributes of a *match* element;
- *Direct Match Links*: are variables typed by labelled relations of the source metamodel. These variables can assume as values relations having the same label in the input model. A direct match link is always expressed between two match elements;
- *Indirect Match Links*: indirect match links are similar to direct match links, but there may exist a path of containment associations between the matched instances. In the implementation the notion of indirect links captures only acyclic EMF containment associations and as such avoids cycles and infinite amounts of matches over the transitive closure of associations in the input models. In our example, indirect match links are represented in all the transformation rules of layer 'Relations' as dashed arrows between elements of the match models;
- *Backward Links*: backward links connect elements of the match and the apply models. They exist in our example in all transformation rules in the 'Relations' layer, depicted as dashed vertical lines. Backward links are used to refer to elements created in a previous layer in order to use them in the current one. An important characteristic of DSLTrans is that throughout all the layers the source model remains unchanged as a match source. Therefore, the only possibility to reuse elements created from a previous layer is to refer to them using backward links;
- *Negative Conditions*: it is possible to express negative conditions over match elements, backward, direct and indirect match links.

The constructs for building transformation rules' apply patterns are:

- *Apply Elements and Apply Links*: apply elements, as match elements, are variables typed by elements of the source metamodel. Apply elements in a given transformation rule that are not connected to backward links will create elements of the same type in the transformation output. A similar mechanism is used for apply links. These output elements and links will be created as many times as the match

model of the transformation rule is instantiated in the input model. In our example, the 'Station2Male' transformation rule of layer 'Relations' takes instances of *Station* and *Male* (of the 'Gender Language' metamodel) which 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 small language for building the values attributes of apply model elements from references to one or more match model element attributes.

Besides the fact that DSLTrans' transformations are free of contructs that imply unbounded recursion or non-determinism, DSLTrans' transformations are strictly outplace, meaning no changes are allowed to the input model. The output metamodel for a DSLTrans transformation can however be the same as the input metamodel. Also, elements cannot be removed from the output metamodel as the result of applying a DSLTrans rule. This restriction is consistent with the usage of model transformations as translations, as no deletion of output elements is strictly required. This is however not the case when transformations are used to encode operational semantics (simulations) of systems. This illustrates the boundaries of the applicability of DSLTrans and that expressiveness reduction entails a compromise with the class of problems that can be tackled.

3 Verifying Properties of DSLTrans Transformations

In figures 4 and 5 we present two properties we wish to prove or disprove regarding all executions of the transformation presented in figure 2. The property in figure 4 means that "*a model which includes a police station that has both a male and female chief officers will be transformed into a model where the male chief officer will exist in the male set and the female chief officer will exist in the female set*". This is something we expect will always hold in our transformation. The property in figure 5 mans that "*any model which includes female officer will be transformed into a model where that female officer will always supervise another female officer*", which is something that we expect will hold for our transformation sometimes, but not always.

As previously mentioned, the properties we are interested in proving are precondition-postcondition axioms. Those preconditions and postconditions are constraints on the input and output models of the DSLTrans transformation being analysed. Preconditions and postcondition constraints are expressed as patterns, primarily as is done respectively in the *MatchModel* and *ApplyModel* patterns of DSLTrans transformation rules. Preconditions use the same pattern language as the *MatchModel* part of DSLTrans rules, involving 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* part of DSLTrans rules – they involve patterns over the transitive closure of containment links in input models. Postconditions also use the same patterns language as the *ApplyModel* patterns of DSLTrans transformation rules, with the additional possibility of also expressing indirect links for patterns involving the transitive closure of containment links in output models. Backward links can also be used in properties to

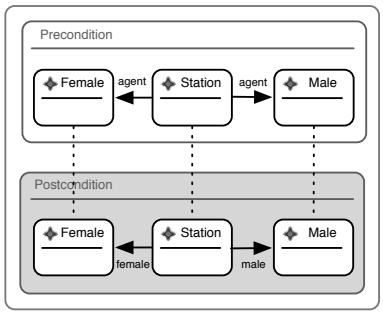

Fig. 4: Police Station Transformation Property 1

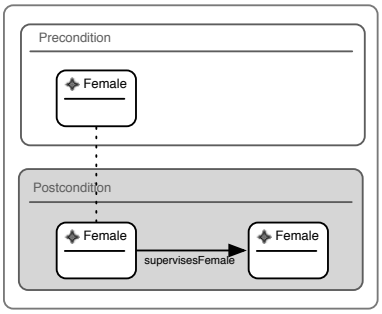

Fig. 5: Police Station Transformation Property 2

impose traceability relations between precondition and postcondition elements. A formal definition of our property language can be found in [5].

In [10] Narayanan and Karsai describe well the nature of these model syntax relation properties, which they call called *structural correspondence rules* in their work. As the authors claim, "*given the correspondence conditions are specified in terms of simple queries on the model around previously chosen context (metamodel) nodes, we expect that they will be easier to specify, and thus more reliable than the transformation itself*".

In what follows we will describe the tool we have built the prove or disprove such properties. Proofs are built relying only the rules of the DSLTrans transformation we are analysing and as such are valid for all input models. Thus, if our prover replies *yes*, then the precondition-postcondition implication expressed in the property will hold for all executions of the DSLTrans transformation under analysis. On the other hand, if the prover replies *no*, then that will mean that there exists at least one exception to the implication in the property. In other words, there exists at least one model in which the precondition of the property holds, but the postcondition does not. A counterxample can be provided in this case, consisting of the sequence of rules that were executed leading to the property being violated.

3.1 Symbolic Execution Construction

In order to explain the concept of symbolic execution of a DSLTrans transformation, let us make an analogy with program symbolic execution as introduced by King in his seminal work "Symbolic Execution and Program Testing" [13]. According to King, a symbolic execution of a program is a set of *path conditions* on that program's input variables. Each path condition describes a traversal of the conditional branching commands of that program. Path conditions are symbolic in the sense that they represent many concrete executions corresponding to (a possibly infinite number of) instantiations of the variables in those conditions.

We can transpose this notion of symbolic execution to DSLTrans model transformations. The analog of an input variable in the model transformation context are meta-

model classes, relations and attributes. As program statements impose constraints on input and output variables during symbolic execution, transformation rules impose conditions on metamodel elements. In fact, when a transformation rule is executed we can assume that the input and output models of the transformation include instances of those constrained metamodel elements. Sets of transformation rules can then be used as the analog of path conditions in the context of model transformations. We must not forget that in a model transformation, rules are implicitly or explicitly scheduled. Such control and/or data dependencies must also be taken into consideration during symbolic execution construction for a model transformation. Such dependencies define which combinations of transformation rules are allowed, similar to what happens with control and data flow in program symbolic execution.

Algorithm 1 Symbolic Execution Generation

```
1:  procedure GENSYMBEXEC(transf)
2:      symbExec = ∅
3:      for curLayer ∈ transf do
4:          if curLayer = firstLayer then
5:              symbExec = P(curLayer)
6:          else
7:              CurLayerSymbExec = P(curLayer)
8:              pathCondsToAdd = ∅
9:              for PC_{l+1} ∈ CurLayerSymbExec do
10:                 if existBackwardLinks(PC_{l+1}) then
11:                     for PC_l ∈ symbExec do
12:                         if ∀backLink ∈ PC_{l+1}, ∃rule ∈ PC_l.backLink ∈ traces(rule) then
13:                             mergedPathCond = PC_l ⊎ PC_{l+1}
14:                             pathCondsToAdd = pathCondsToAdd ∪ mergedPathCond
15:                             if noOvelappingMatchesExist(mergedPathCond) then
16:                                 symbExec = symbExec \ {PC_l}
17:                             end if
18:                         end if
19:                     end for
20:                 else
21:                     pathCondsToAdd = pathCondsToAdd ∪ (PC_l ∪ PC_{l+1})
22:                 end if
23:             end for
24:         end if
25:         symbExec = symbExec ∪ pathCondsToAdd
26:     end for
27:     return symbExec
28: end procedure
```

The construction of a symbolic execution for a DSLTrans transformation is described in algorithm 1. The algorithm starts by processing the transformations' first layer. The symbolic execution of a layer is built as the set of all the possible combinations of that layer's rules (line 5). In mathematical terms it is the powerset of the

considered rules. As previously mentioned, each set of rules in a symbolic execution is a *path condition*. It is important to mention at this point that the fact that in DSLTrans the rules within a layer can be executed in any order with a deterministic result allows us only consider only a fraction of the path conditions that would be necessary if order would be relevant.

Having produced the symbolic execution for the first layer, or any generic layer l, we can now proceed to layer $l+1$ (line 7). As before, we calculate the powerset of the rules in layer $l+1$. However, we now need to understand how each one of these newly built path conditions affects each partial path condition built for layer l. When we analyse a path condition belonging to the powerset of layer $l+1$ (noted in what follows $PC_{l+1} \in \mathcal{P}(l+1)$) against a path condition belonging to the symbolic execution built by the rules of layer l (noted in what follows $PC_l \in \mathcal{P}(l)$), several cases may occur (lines 9-23):

1. If none of the rules in the PC_{l+1} contains backward links, the union of PC_l and PC_{l+1} can be added to the symbolic execution (line 21). The union is built by adding the rules in PC_{l+1} to the rules in PC_l. In terms of symbolic execution this means that, given there is no dependency between the rules in PC_{l+1} and the rules of PC_l, the rules in in PC_{l+1} may execute or not depending on the input model. Because symbolic execution is independent of the input model, we need to consider both possibilities by keeping in the symbolic execution both PC_l and $PC_l \cup PC_{l+1}$;

2. If the rules in PC_{l+1} include backward links, then we need to analyse whether those backward links correspond to traces between match and apply elements generated by rules in PC_l (line 12). If this is not the case then the conditions for at least one of the rules from PC_{l+1} to execute is not satisfied and PC_{l+1} cannot be added to the symbolic execution. Figure 6 illustrates this case, given that the backward link the Station2Female rule in PC_{l+1} connecting the match element *Female* to the apply element *Female* does not have a corresponding trace[2] in PC_l;

3. If the rules in PC_{l+1} include backward links and all those backward links correspond to traces between match and apply elements generated by rules in PC_l then, as in case 1, we can add to the symbolic execution a new path condition including PC_l and PC_{l+1}. An example of this case can be observed in figure 7. However, in this case the union of PC_l and PC_{l+1} is performed differently than in case 1: all rules from PC_{l+1} containing backward links are merged with the rules from PC_{l+1} where the traces corresponding to the backward links were generated by using operator $\uplus$ (line 13). Additionally, PC_l is removed from the symbolic execution (line 16). This is because, given all elements necessary for rules of PC_{l+1} including backward links were generated by the rules of PC_l, the rules from PC_{l+1} with backward links necessarily execute. As such, PC_l can no longer exist on its own in the symbolic execution;

[2] Note that in figure 6, in order to make explicit the traces between apply elements generated from match elements in rule Station2Station, we added to the original rule thick dashed lines to connect those elements. The same principle applies to figures 7 and 8.

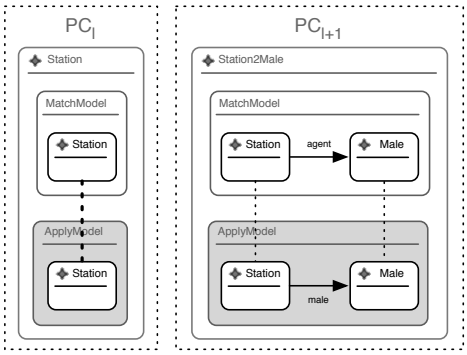

Fig. 6: Non Mergeable Symbolic States

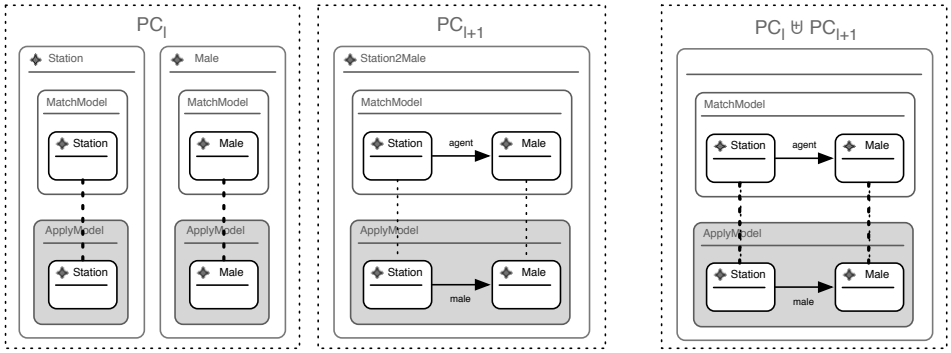

Fig. 7: Mergeable Symbolic States

4. A variation of case 3 is the case when more than one backward link from the same rule in PC_{l+1} is matched over the same trace of a rule from PC_l (line 15). An example of this case can be observed in figure 8. In this case, additionally to what happens in case 3, PC_l is kept in the symbolic execution. This is due to our abstraction over the number of matches: referring to our example in figure 8, because we are not sure if the Female2Female rule applied more than once, we cannot decide whether rule Female2Female be executed or not and both cases need to be kept in the symbolic execution.

3.2 Property Proof

After the symbolic execution construction is finished, property proof can start. For each built path condition of the completed symbolic execution, property proof is performed as shown in algorithm 2. The algorithm behaves as follows:

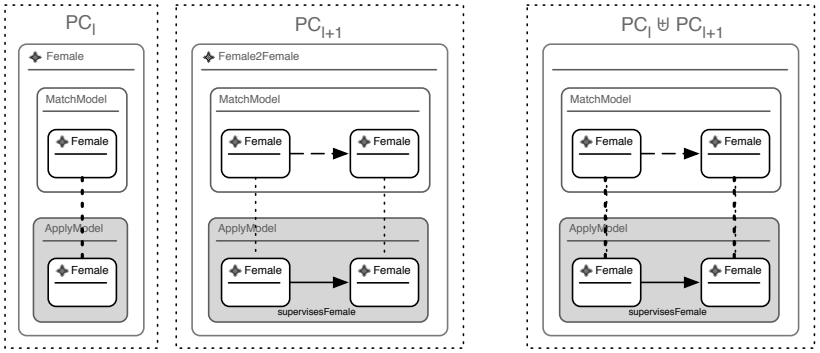

Fig. 8: Mergeable Symbolic States with Repetition

A path condition is taken from the symbolic execution (line 2). If no more path conditions exist, the property holds (line 13). A first check is done to decide if the path condition under consideration has all the metamodel match elements expressed in the property's precondition (line 3). Because we do not know whether two match elements of the same type occurring in two different rules in the same path condition consume the same or different instances in a concrete input model, we need to consider two cases:

- the case where those two distinct match elements from different rules consume two different instances of that type in the input model;
- the case where those two match elements consume the same instance in the input model.

This is achieved by building for a given path condition, all the possibilities of collapsed elements of the same type belonging to different rules (line 4). We then iterate over the path conditions resulting from the collapse operation. For each of those path conditions we check whether the precondition part of the property is a subgraph of the path condition (line 5), in which case we check if the whole property is a subgraph of the collapsed path condition (line 6). If this is not the case then the property does not hold and the path condition itself serves as counterexample for the property (line 7).

We will exemplify the *Collapse* algorithm in line 4 of algorithm 2 on the path condition in figure 9. The algorithm starts by finding all the pairs of elements of the same type in the path condition's rules (line 2). In figure 9 the only available pair of *Station* elements is highlighted by dashed ellipses.

The algorithm then goes on to moving all the links pointing to one of the chosen match elements in the pair to the other element in the pair and deleting the stripped match element (line 5). Figure 10 shows the result of applying this step to our example. Note that the choice of which of the pair's match element will hold all links belonging to both elements is non deterministic.

At this point of the algorithm, if the two merged match elements are connected by backward links and both those backward links connect to apply elements having the same type (line 6), then those apply elements need to be merged in the same way

Algorithm 2 Prove

```
1:  procedure PROVE(symbExec, property)
2:      for PC ∈ symbExec do
3:          if checkPreconditionElements(PC, property) then
4:              for collapsedPC ∈ Collapse(PC) do
5:                  if checkPrecondition(PC, property) then
6:                      if not checkPrePostCondition(PC, property) then
7:                          return PC
8:                      end if
9:                  end if
10:             end for
11:         end if
12:     end for
13:     return True
14: end procedure
```

Algorithm 3 Collapse

```
1:  procedure COLLAPSE(PC)
2:      collapsableMatchPairSet = getSameTypeDiffRulesMatchPairs(PC)
3:      collapsedPCSet = ∅
4:      for matchPair ∈ collapsablePairSet do
5:          PC' = moveAllLinks(matchPair, PC)
6:          if matchPair = (m1, m2) ∧ ∃(a1, a2). backwardLink(m1, a1) ∧
            backwardLink(m2, a2) ∧ sameType(a1, a2) then
7:              PC' = moveAllLinks((a1, a2), PC')
8:          end if
9:          collapsedPCSet = collapsedPCSet ∪ PC' ∪ Collapse(PC')
10:     end for
11:     return collapsedPCSet
12: end procedure
```

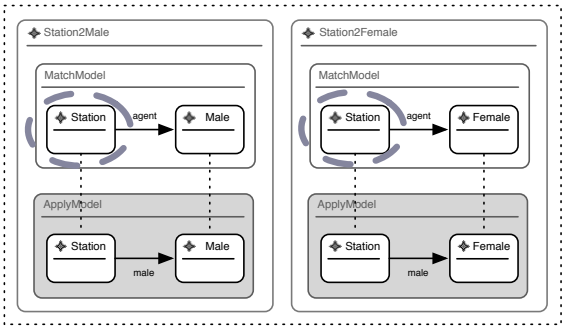

Fig. 9: Collapse algorithm: locating two Match elements of the same type

as the match ones were (line 7). The reason for this is that backward links refer to transformation steps that were previously executed, and as such if more than one rule refers to a previous step, then that previous step is necessarily the same. In our example in figure 10, both backward links connecting *Station* match and apply elements refer to rule *Station2Station* in the first layer of the police station transformation in figure 2. We have thus highlighted in figure 10 the two apply elements to be merged.

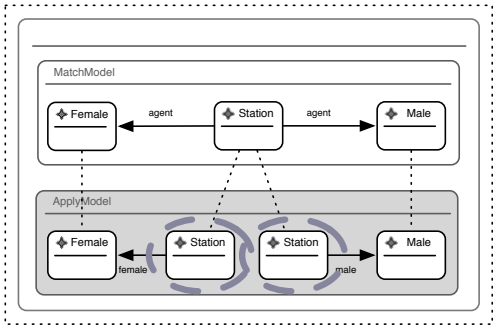

Fig. 10: Collapse algorithm: merging two Match elements and locating two Apply elements of the same type

If apply elements exist and they are merged, the collapse step is then complete. For our example the result of the collapse step is depicted in figure 11. Algorithm 3 will then proceed by adding to the set of collapsed path conditions to return: (i) the result of the current collapse step; (ii) the recursive result of collapsing the rules remaining from the current collapse step (line 9). The algorithm will then loop over the remaining pairs of match elements having the same type until no more pairs exist and when that happens return all produced collapsed path conditions (line 11).

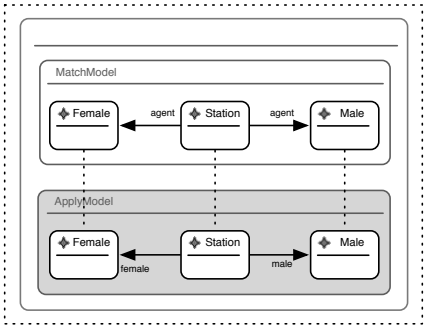

Fig. 11: Collapse algorithm: merging two Apply elements

4 Property Prover Architecture

In the text that follows we introduce the architecture of the tool we are currently developing to prove properties of DSLTrans transformations. The tool construction and operation follows MDE principles in the sense that all artifacts are explicitly modelled by the appropriate metamodels and computations are performed using model transformations.

In order to build a property prover for a given DSLTrans transformation several steps are required. In a complete tool all these steps can and should be automated using Higher Order Transformations (HOT). However, given our first goal was to demonstrate that the approach scales to prove properties of usable transformations, all the steps that follow have been manually performed for the *police station* transformation example. Figure 12 shows the several higher order transformations needed by our framework, along with other required artifacts such as metamodels and models. The numbered higher order transformations in figure 12 are described in the text that follows:

1. **Generate the *abstract rule metamodel***: this HOT takes as input the source and target metamodels of the Transformation Under Analysis (TUA) and returns a metamodel in which an abstract form of the transformation rules can be written. Such a metamodel for the police station transformation can be observed in figure 13. The purpose of these abstract rules is to be the basic building blocks during symbolic state space construction;

2. **Generate the *property metamodel***: this HOT builds the metamodel which can be used to express properties about the TUA. It takes as input the source and target metamodels for the TUA and returns the language in which properties are written. For example, the properties in figures 4 and 5 are written using the *property metamodel* generated for the Police Station transformation;

3. **Generate the *symbolic state space construction rules***: this HOT builds a set of models corresponding to the abstract form of the TUA rules to be used during the symbolic state space construction. As depicted in figure 12, the rules generated by this HOT are instances of the *abstract rule metamodel*. As an example, the rules in the path conditions in figures 6, 7 and 8 are *symbolic state space construction rules*;

4. **Generate the *backward link match transformation***: builds the query transformation responsible for checking 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*. The rules in this transformation is used to evaluate the condition in line 12 of algorithm 1 (Symbolic Execution Generation);

5. **Generate the *collapse transformation***: this HOT takes as input the *abstract rule metamodel* generated in step 1 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*. The rules in the *collapse transformation* are used in lines 2, 5, 6 and 7 of algorithm 3 (Collapse);

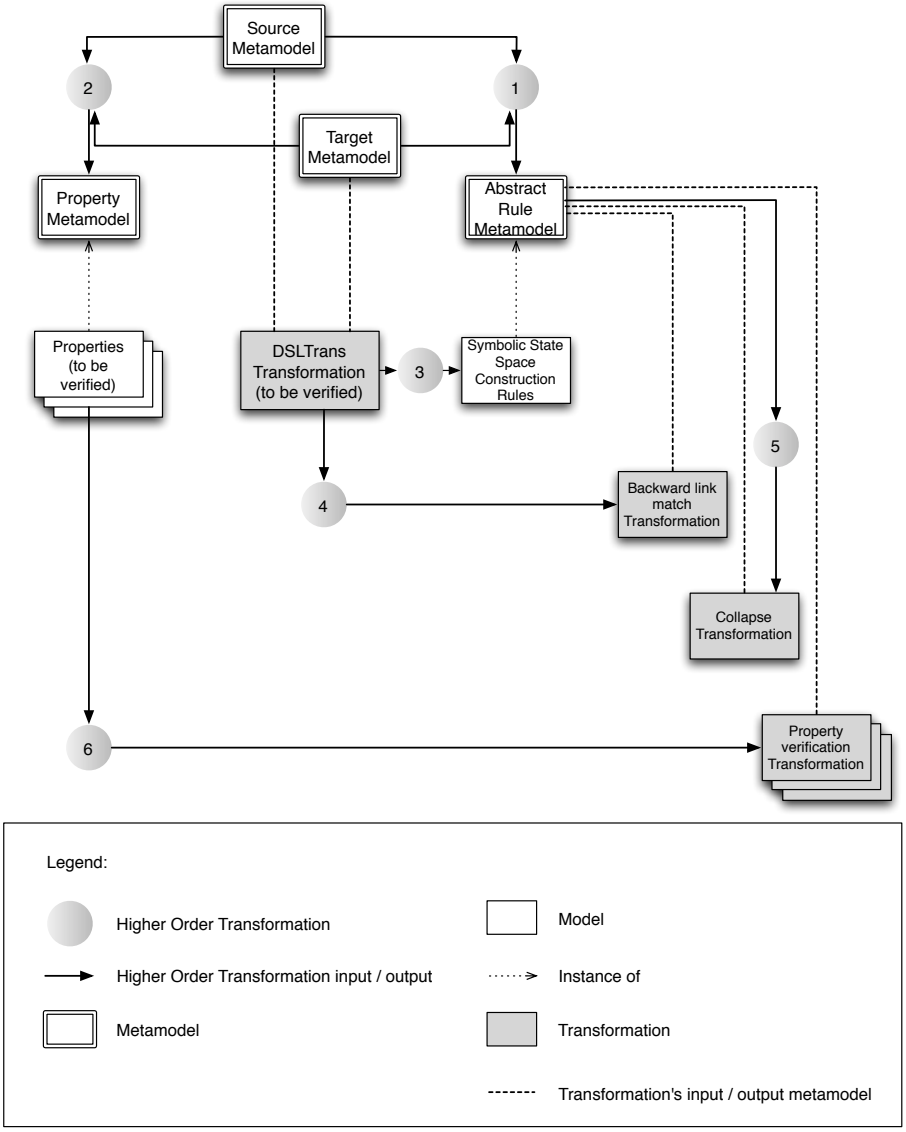

Fig. 12: Verification Tool Architecture

6. **Generate the *property verification transformation***: this HOT generates three query transformations for each property to be verified:
   - a first query transformation that checks whether the model elements present in the property are present in a symbolic state;

– a second query transformations that checks whether the match part of the property is a subgraph of the match part of a possibly collapsed symbolic state;
– a third query transformation that checks whether the whole property is a subgraph of a possibly collapsed symbolic state.

All query transformations have as source metamodel the *abstract rule metamodel*. The rules in a given *property verification transformation* are used in lines 3, 5 and 6 of algorithm 2 (Prove).

5 Enabling Technology

As introduced in the previous two chapters, a large amount of the work necessary to prove the properties of DSLTrans transformations we have introduced relies on graph matching and/or graph rewriting. As described in section 4, in our tool model transformations are used at two levels. First, during a 'compilation' step, all the necessary metamodels and transformations required to perform analysis on a given DSLTrans transformation are produced by higher order transformations. The six higher order transformations defined in section 4 are independent of the TUA and are part of our toolset. Then, the model transformations generated during the first step are used in the Symbolic Execution Generation, Prove and Collapse algorithms to build the symbolic state space for the TUA and prove or disprove properties of interest. These model transformations are generated for each TUA and additionally for each property to be proved for each TUA.

When deciding on a model transformation framework to build our verification tool for DSLTrans transformations it became clear that we required very detailed control over the behavior of those model transformations and the way in which they are scheduled such that our the algorithms in section 3 can be built. Moreover, support for higher order transformations in necessary and as such a transformation language with an explicit metamodel is required.

In order to build our tool we have chosen the T-Core framework introduced by Syriani and Vangheluwe in [14]. T-Core is a set of primitive model transformation blocks that can be used to replicate the behavior of existing transformation languages (e.g. in order to compare their expressiveness and provide a framework for interoperability) or to build new model transformation languages. The framework includes five main primitive transformation blocks that exchange models and transformation information in messages called *packets*. Those blocks are: the *Matcher*, that finds matches of a given pre-condition pattern within a model by running an efficient combination of the Ullmann's and VF2's subgraph isomorphism algorithm and collects those matches in a *packet*; the *Iterator* which allows selecting the next matched submodel from the set of matches gathered in a *packet* such that a part of the model to be changed can locked on; the *Rewriter* consumes a matched subgraph from a model in a packet and changes the model according to a given post-condition pattern; the *Rollbacker* which allows checkpointing and restoring *packets* such that backtracking can be achieved; the Resolver for solving potential conflicts between matches and rewritings. An additional construct called the *composer* is used to encapsulate compositions of the five primitive transformation blocks described above. The goal of the encapsulation mechanism is

that complex transformation blocks such as for example *querying*, *rewriting one random match* or *rewriting all matches found* can be seamlessly created from the simplest transformation blocks.

Note that the transformation primitives described above execute transformations on models which are metamodel instances. Matching precondition patterns and rewriting postcondition patterns can only occur if the pre- and post-condition patterns are generated from the same metamodel the models being treated in order to insure coherence.

6 Implementation and Performance Experiments

As mentioned in section 4, in order to conduct our experiments with the technique presented in this paper we have implemented manually the higher order transformations in figure 12 required to build the symbolic state space and do the property verification for our police station case study transformation. This implied developing all the required metamodels, models and transformation rules for the police station transformation as described in figure 12. In order to do so we have used the AToM^3 metamodelling environment [15] in which all these artifacts can be built. In particular we have constructed:

- both the metamodels for the *Abstract Rule Metamodel* and the *property metamodel*. We depict in figure 13 the abstract rule metamodel for the police station transformation as displayed in the AToM^3 tool. Note that all constructs in the metamodel reflect the structure of DSLTrans' rules, including Match and Apply Models, Match and Apply Model Elements and the possible links between these entities. However, the *MetaModelElement_S* and *MetamodelElement_T* are superclasses of all possible types present in the *source* and *target* metamodels respectively, in our case the *Station*, *Male* and *Female* types. Type names are distinguished by a *classType String* attribute in the *MetaModelElement* and relation names are distinguished by an *associationType String* attribute in the *directLink* association classes. Note that, although the *abstract rule metamodel* in figure 13 is specific to the police station transformation, it defines a template for such metamodels that can be built for any input and output metamodels of a DSLTrans transformation by a HOT;
- the *Abstract Transformation Rules* which are 7 models, one per transformation rule in the police station transformation in figure 2;
- the *Backward Link Query Transformation* which consists of 8 transformation rules, one per each subgraph connected by backward links in each of the rules in the second layer of the police station transformation in figure 2;
- the *Collapse Transformation*, consisting of 16 transformation rules that form the building blocks of algorithm 3;
- the *Property Verification Query Transformation*, which consists of 3 transformation rules per property to be proved for the police station transformation. For our experimental purposes we expressed the properties to be proved directly as transformation rules and bypassed the property expression as an instance of the property metamodel.

Our prototype was developed using a mix of Python and T-Core and can be found in [16]. Given T-Core is built as a Python library, T-Core primitives can be embedded in

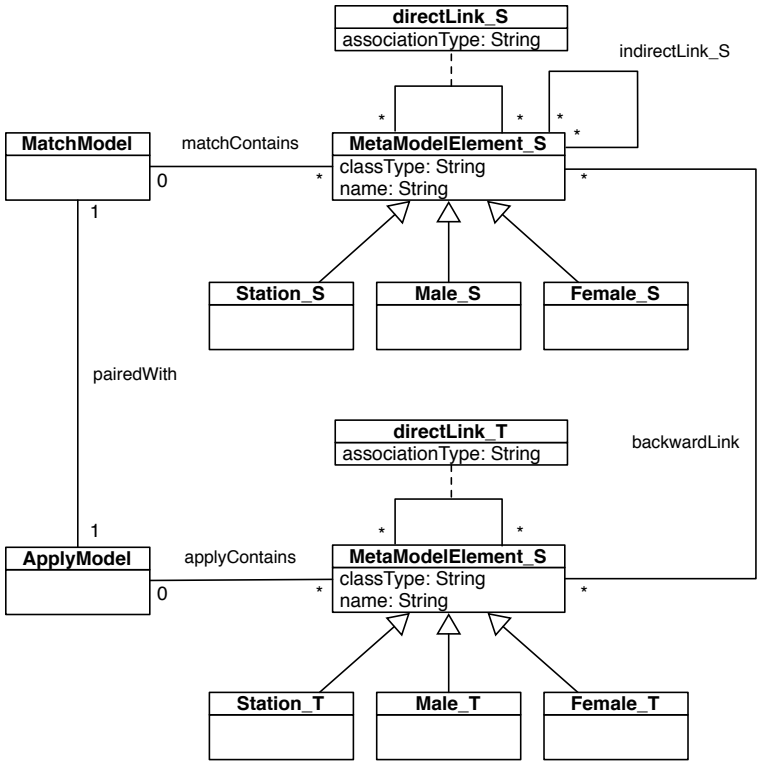

Fig. 13: Abstract Rule Metamodel for the Police Station Transformation

python code such that the required scheduling to build the algorithms described in section 3 can be achieved. These primitives are initialized with the pre- and post-condition patterns (the transformation rules) built by higher order transformations 4, 5 and 6 in figure 12. The initialized transformation primitives then act on the *Abstract Transformation Rules* built by higher order transformation 3 in figure 12 and are scheduled using Python code according to algorithms 1, 2 and 3.

Given that many similar situations have to be investigated during symbolic execution construction and property proof, memoisation was used whenever possible to avoid isomorphic graph matching and rewrite operations. In what concerns algorithm 1, we have used pointers to rules instead of copies of rules to build each path condition. Caches were used to accelerate the repetitive backward link match transformations and the *mergeRulesOverBackwardLinks* operations between rules of different layers. For algorithm 3, instead of the recursive strategy explained in section 3 we have implemented a method to build the set of collapsed path conditions for a given path condition incrementally. The strategy involved starting by collapsing rules in a path condition two by two, then collapsing again the results with remaining rules, and so on. This solution works because the collapse operation is associative. In this fashion no repeated col-

lapsed path conditions are built and intermediate results of collapsing several rules can be cached to be used when collapsing the same rules in different path conditions. It is however the case that, if no or few collapse operations are required for a path condition, the algorithm still attempts to collapse rules in this fashion. This may result in overhead as compared to merging all the rules and running recursive algorithm 3 directly.

For property proof we have also implemented a strategy to avoid checking path conditions of the symbolic execution where the property is sure to hold. The strategy is based on the fact that 1) if a path condition PC' contains the same rules as a path condition PC where the property has already been checked successfully and 2) no other elements influencing the property exist in PC', then the property still holds for PC'.

In order to understand the performance of our approach we have used our tool to check the properties in figures 4 and 5. Although are many variables that need to be taken into consideration when performing such a analysis, we have started by the most basic variable that may intuitively influence the performance of our approach: the number of rules in the transformation under analysis.

Our experiment is based on the transformation we have presented in figure 2. In order to have more than the 7 rules in the original Police Station transformation we have replicated those 7 rules four times, in order to reach a maximum of 28 rules distributed by 5 layers as shown in figure 14. Note that in figure 14 and for clarity reasons we abbreviated the match model and apply model elements *Station*, *Male* and *Female* to *S*, *M* and *F* respectively. Additionally, to distinguish between the replicated versions of each rule we have added an index to each match and apply element name. In this extended transformation different indexes correspond to different source and target metamodel elements. As an example, the *S1* match element matches different model elements than match element *S2*.

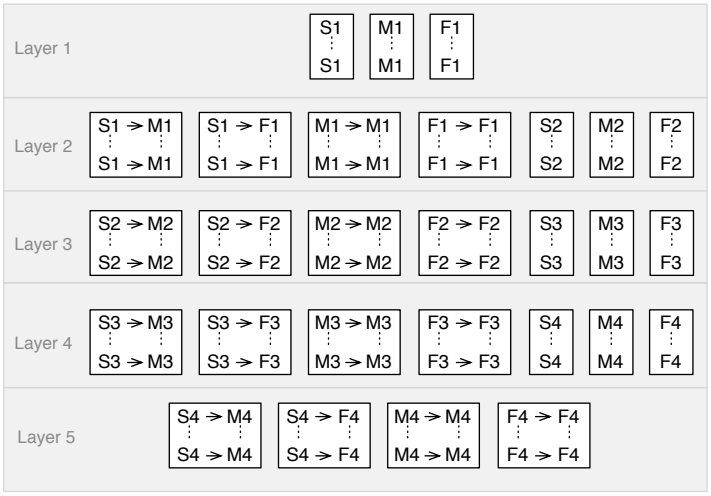

Fig. 14: Replicated Police Station Transformation for Performance Tests

# of rules	3	5	7	10	12	14
# of path conditions	8	14	31	269	337	1051
symbolic execution construction time (sec)	2.3×10^{-5}	0.12	0.25	0.27	0.40	0.93
used memory (Kb)	0.07	0.09	0.17	1.08	1.41	4.40
Prop. 1, repl. 1 (sec)	-	0.11	0.68	1.80	2.21	6.97
Prop. 1, repl. 2 (sec)	-	-	-	-	1.41	7.06
Prop. 1, repl. 3 (sec)	-	-	-	-	-	-
Prop. 2, repl. 1 (sec)	-	1.8×10^{-3}	1.8×10^{-3}	1.5×10^{-3}	1.7×10^{-3}	1.6×10^{-3}
Prop. 2, repl. 2 (sec)	-	-	-	0.04	0.04	0.04
Prop. 2, repl. 3 (sec)	-	-	-	-	-	-
Prop. 1, repl. 1 $\wedge$ Prop. 1, repl. 3 (sec)	-	-	-	-	-	-
Prop. 2, repl. 1 $\wedge$ Prop. 2, repl. 3 (sec)	-	-	-	-	-	-
# of rules	17	19	21	24	26	28
# of path conditions	9122	11428	35641	309341	387541	1208641
symbolic execution construction time (sec)	4.56	9.88	53.27	1222.11	3144.98	30513.64
used memory (Kb)	38.01	48.16	139.35	1307.66	1655.05	4777.00
Prop. 1, repl. 1 (sec)	66.42	88.57	320.00	-	-	-
Prop. 1, repl. 2 (sec)	66.87	89.87	347.01	-	-	-
Prop. 1, repl. 3 (sec)	-	78.92	323.50	-	-	-
Prop. 2, repl. 1 (sec	1.5×10^{-3}	1.6×10^{-3}	1.6×10^{-3}	-	-	-
Prop. 2, repl. 2 (sec)	0.04	0.04	0.04	-	-	-
Prop. 2, repl. 3 (sec)	-	4.41	4.39	-	-	-
Prop. 1, repl. 1 $\wedge$ Prop. 1, repl. 3 (sec)	-	109.41	649.01	-	-	-
Prop. 2, repl. 1 $\wedge$ Prop. 2, repl. 3 (sec)	-	5.22	5.03	-	-	-

Table 1: Performance Results for the Proof of Properties of the Police Station Transformation

In table 1 we present some performance results for the Police Station transformation. The results presented in table 1 were obtained using a 2.2 GHz Intel Core i7 machine with 8GB of DDR3 memory running Ubuntu 11.10 and Python 2.7. For each measurement involving time we repeated the given experiment three times and calculated the final result as the average of the three experiment results. The code we used in our experiments can be found under [16].

The first line of the table 1 shows the number of rules for each part of the experiment. The rules that are involved in each transformation in the experiment can be deduced from this number by counting the rules horizontally starting from the top left corner of figure 14. For example 5 rules corresponds to the three first rules of layer 1 plus the two

first rules of layer 2; 7 rules correspond to the first three rules of layer 1 plus the four first rules of layer 2; and so on.

Lines 2 and 3 in table 1 present respectively the number of path conditions and the computation time for the symbolic execution construction for the given amount of rules. Both the number of path conditions and the time for building the symbolic executions raise steeply with the number of rules, but for our example it is reasonable to build symbolic executions and prove properties for up to 21 rules. In order to test the limits of our approach we have tried building symbolic executions up until 28 rules. In this case the number of built path conditions is over 1 million and the time to do so over 8 hours. We have not attempted to prove properties of transformations with more than 21 rules due to the long running times required. From Line 4 in table 1 we can see that memory consumption is very modest, even when the symbolic execution is composed of more than one million path conditions. This is due to the fact that the actual number of DSLTrans rules used for path condition construction is very low and we only use pointers to the actual rules in memory.

Lines 5 through 7 of table 1 present the times to prove property 1 of the Police Station transformation (see figure 4). Note that the property holds for any amount of rules. We have replicated property 4 three times, one time per each of the replicated set of rules for the Police Station transformation. It is clear from table 1 that the time to prove any of the replicas of the properties increases with the number of rules. This is due to the fact that property 4 holds, and as such the whole set of path conditions for the symbolic execution needs to be checked. For that reason proof time naturally increases with the size of the symbolic execution. However, proof time does not change significantly for the several replicas of the property. This is due to the fact that, given rules are replicated, the proof computations for each property replica are similar. From this fact we can deduce that the position of the rules in the transformation affected by the property under proof does not affect proof time.

Lines 8 through 10 of table 1 present the times to disprove property 2 of the Police Station transformation (see figure 5). The property does not hold for any amount of rules. For all the replicas of property 5 the proof times were constant. This is due to the fact that, given the property does not hold, the proof algorithm can stop as soon as a counterexample is found. The proof times increase for each replica of the property given that replicas further down the table refer to rules that appear in later layers of the Police Station transformation. As such the proof algorithm reaches the path conditions involving those rules later and requires more time.

Line 11 of table 1 presents the time to prove a property which is a conjunction of replicas 1 and 3 of the property in figure 4. The conjunction is achieved by merging the two replicas is the same graph. The property holds and the proof time is higher than for the individual smaller properties on lines 4 and 6. This is due to the fact that, because the property involves more match and apply elements, the subgraph isomorphim checks for property proof require more time than either of the two replicas that compose it. Exactly the same principle applies to the property in line 12, which is the conjunction of replicas 1 and 3 of the property in figure 5. Due to the size of the property, the proof time is higher than the proof time for the individual replicas. As expected for properties that do not hold, proof time across line 11 is constant as the number of rules increases.

Finally, one last observation on the values in table 1 is the fact that when rules without backward links are added to the transformation (the jumps from 7 to 10, 14 to 17 and 21 to 24 rules) both the number of produced path conditions and the used memory increases around tenfold. Comparatively, when rules with backward links are added to the transformation (the jumps from 3 to 7, 10 to 14 and 17 to 21 rules) both the number of path conditions and the used memory only increases around five times. This is despite the fact that we increase rules with backward links by sets of 4 rules, whereas we increase rules without backward links by sets of 3 rules. The reason for this difference is the fact that: during symbolic execution construction a rule with backward links may simply not be executable for a given path condition or can be merged with a previous path conditions – in both these cases no new path conditions are added to the symbolic execution; however a rule without backward links necessarily generates twice the path conditions already generated by the symbolic execution algorithm as it may always be executed.

7 Discussion of the Results and Contributions

The contributions of the work presented in this paper are the following:

- The algorithms to build the symbolic execution for a given DSLTrans' model transformation and prove structural properties of that transformation based on the constructed symbolic execution. This work materialises our proposal originally presented in [5];
- A prototype implementation of those algorithms based on the T-Core model transformation framework [14] is described. Based on our running example we have performed a detailed performance study and have shown that our technique can easily scale up to a transformations including around 21 transformation rules which, given our experience with DSLTrans, are transformations of a useful size to specify real world transformations. We have performed experiments with up to 28 rules, at which point the time taken by the required computations became unreasonably high;
- The architecture of tool to allow for performing the analysis of any DSLTrans transformation is provided. We have produced by hand the artifacts necessary for building the symbolic execution for our running example and proving its structural properties. Higher order transformations can be used to automate the production of such artifacts for an arbitrary DSLTrans transformation;
- We demonstrate a symbolic execution can be practically built and exploited for a model transformation specification. To the best of our knowledge of the literature of the domain, this hasn't been attempted yet for a model transformation language. In order to build such a symbolic execution we base our proposal on the fact that DSLTrans is a language that guarantees by construction both *termination* and *confluence* of all specifiable model transformations. An interesting corollary of our experiments is that model transformations are themselves a useful tool in the construction of proofs of properties of model transformations.

The performance results we have presented in this paper are promising, but further experiments with other transformations need to be done in order to assess the real performance of our approach. In fact, depending on the size of the rules in the considered model transformation, on rule distribution among layers, if those rules involve many backward links or not and whether many elements of the same type exist scattered by different rules or not (implying many collapse operation), we expect that the number of rules that can be tackled by our approach may vary substantially. Also it became clear from section 6 that proof time is high as compared to symbolic execution construction. Our results show that, when the property holds, the proof time is several times larger than the time it takes to build the symbolic execution. We believe the proof time may be considerably reduced by concentrating only on the symbolic execution states that contain rules that are affected by the property to prove rather than checking the final symbolic execution states one by one as we do now.

Another point that needs to be further developed in our approach is the property language. In this paper we have concentrated on building the algorithms that allow symbolic execution construction and property proof, but have left the property language in a relatively basic state, being that for the time being it allows essentially to express what is expressible in transformation rules (including statements about multiples of instances of elements of the same type) and transitive containment connections at the *Postcondition* part of the property. We believe that the current property language can already be very useful to prove many relevant properties of practical transformations, but have not studied its full range yet. Regarding the possible extensions of the property language, inspiration can be drawn from several proposals in the literature [10, 11, 17, 12], further explained in section 8. One of the natural extensions of our property language would be the possibility to express conditions over the attributes of the elements in the properties, which for the time being we do not address. During the symbolic execution construction such conditions will have to be addressed symbolically, which adds an additional challenge to DSLTrans' symbolic execution construction. Negative links (associations, indirect links and backward links) are also part of our future tasks, both at the level of the symbolic execution construction and of the property language.

8 Related Work

In order to analyse the work in the literature that is close to our proposal, we will make use of the study on the formal verification of model transformations proposed in [3]. The study uses three dimensions to classify the analysis of model transformations. The dimensions are: 1) the *kind of transformations* considered; 2) the *properties* of transformations that can be checked; and 3) the *verification technique* used.

In what concerns the *kind of transformations* considered, DSLTrans is a graph based transformation language and as such shares its principles with languages such as AGG [18], AToM^3 [19], VIATRA2 [20], ATL [21] or VTMS [22]. As mentioned previously, DSLTrans' transformation are *terminating* and *confluent* by construction. This is achieved by expressiveness reduction which means that constructs which imply unbounded recursion or non-determinism are avoided. DSLTrans is, to the best of our

knowledge, the only graph based transformation language where these properties are enforced by construction.

It is recognized in the literature that *termination* and *confluence* are important properties of model transformations. This is so because transformations that have such properties are easier to understand and analyse. However, given that termination is undecidable for graph based transformation languages [23], termination criteria and techniques for analysing such criteria on transformations written in graph based transformation languages [24–28] have been proposed to alleviate this problem. Confluence is also undecidable for graph based transformation languages [29]. As for termination, several confluence criteria and corresponding analysis techniques have been proposed in the literature [30, 28, 31, 32].

Regarding the *properties* of transformations that can be checked, according to the classification in [3] the technique presented in this paper deals with properties that can be regarded as *model syntax relations*. Such properties of a model transformation have to do with the fact that certain elements, or structures, of the input model are necessarily transformed into other elements, or structures, of the output model.

As early as 2002 Akehurst and Kent have introduced a set of structural relations between the metamodels of the abstract syntax, concrete syntax and semantics domain of a fragment of the UML [9]. Although they do not use such relations as properties of model transformations, their text introduces the notion of structural relations between a source and a target metamodel for a transformation. Later, in 2007, Narayanan and Karsai propose verifying model transformations by structural correspondence [10]. In their approach structural correspondences are defined as precondition-postcondition axioms. Such that the axioms provide an additional level of specification of the transformation, they are written independently from the transformation rules and are predicate logic formulas relying solely on a pair of the transformation's input and output model objects and attributes. The verification of whether such predicates hold is achieved by relying on so-called cross links (also named *traceability links* in [3]) that are built between the elements of the input and output transformation model during the transformation's execution. Although our proposal follows the same basic idea as the work of Narayanan and Karsai, there is one essential difference. Narayanan and Karsai's technique is focused on showing that precondition-postcondition axioms hold for one execution of a model transformation, involving one input and its corresponding output model. Thus, according to [3] the technique is *transformation dependent* and *input dependent*. With our proposal we aim at proving structural correspondences for all executions of a model transformation, while basing the construction of the properties (or precondition-postcondition axioms, using the vocabulary in [10]) on the source and target metamodel structures. Our approach is thus *transformation dependent* but *input independent* and aims at achieving the proof of the same kind of properties as Narayanan and Karsai propose, but one meta-level above.

In 2009 [11] Cariou *et al.* study the use of OCL contracts in the verification of model transformations. The approach is also *transformation dependent* and *input dependent* in the sense that it requires and input and an output model of the transformation. However,

the authors provide a good account how to build OCL contracts for model transformations and show how to verify those contracts for endogenous transformations.

Aztalos, Lengyel and Levendovszky have published in 2010 their approach to the verification of model transformations [17]. They propose an assertion language that allows making structural statements about models at a given point of the execution of the transformation and also statements about the transformation steps themselves. The authors' technique applies to transformations written in the VTMS transformation language [22]. The technique consists of transforming VTMS transformation rules and verification assertions into Prolog predicates such that deduction rules encoding VTMS's and the assertion language's semantics can be used on automated Prolog proofs to check whether those assertions hold or not. The approach resembles ours in the sense that the technique is also *transformation dependent* but *input independent* (the authors call their technique *offline*). The artifacts used in the proofs are also generated from the transformation and the properties to be proved. While it is foreseeable that our *model syntax relations* properties might be expressed by the assertion language proposed by Aztalos *et al.*, the authors provide no account of the scalability of their approach. They mention mention however that the fact that their approach is based on the generic SWI-Prolog inference engine can be a performance bottleneck or induce non-terminating computations. They foresee that a specialised reasoning system might be necessary for their approach to scale.

More recently in 2012 Guerra *et al.* have proposed a technique for the automated verification of model transformations based on visual contracts [12]. The paper describes a rich and well studied language for describing syntactic relations between input and output models. Contrary to our approach, Guerra *et al.* follow a testing approach in the implementation of their: they use the visual contracts as a means to specify an oracle that can decide whether the result of executing a model transformation is correct or not. A transformation can thus be tested by a batch of existing input models. The approach is *transformation dependent* and *input independent* and is independent of the transformation language used, which is a feature that we have not found elsewhere in the literature. On the other hand the verification technique used by Guerra *et al.* differs fundamentally from ours since, within our abstraction over the number of elements of the same type present in the model, our approach is exhaustive and can provide correctness proofs, where the author's approach is aimed at increasing the level of confidence without providing a definitive proof.

Also in 2012 Büttner *at al.* have published their work on the verification of ATL transformations [33, 34]. In [33] the authors translate ATL transformations and their semantics into transformation models in Alloy. They then use Alloy's model finder to search for the negation of a given property, expressed as an OCL constraint, that should hold. As the authors mention, Alloy performs bounded verification and as such it does not guarantee that a counterexample is found if it exists. In [34] Büttner *at al.* aim at proving model syntax relation properties of ATL transformations expressed as precondition-postcondition OCL constraints. In order to do so the authors provide and use an axiomatisation of ATL's semantics in first order logic. Verification of a given model transformation is achieved by using a HOT to transform the TUA into additional first order logic axioms. Off-the-shelf SMT solvers such as Z3 and Yices are then

used to check whether the precondition-postcondition OCL constraints hold. The approach in [34] comes very close to ours as the authors aim at proving the same type of properties in a model independent fashion and can do so exhaustively by using mathematical proofs at an appropriate level of abstraction, which can be seen as symbolic. The are several differences with our approach: the authors' proofs may require human assistance, depending on the used SAT solver; despite the fact that Büttner *at al.* do treat constraints on object attributes, which we do not do, their scalability results are presented for a small (6 rule) transformation; contrarily to DSLTrans, ATL does not have explicit formal semantics and because of that Büttner *at al.*'s axiomatisation of ATL's semantics is tentative. More generally, while the authors' approach requires an intermediate logic representation of the transformation under analysis, our symbolic approach deals directly with transformation rules. This feature can ease the interpretation of analysis results such as counterexamples and be in general less error prone due to the absence of an indirection layer mapping transformation concepts to concepts in the chosen logic. It is interesting to notice that, as us, Büttner *at al.* have chosen *expressiveness reduction* as a means to work with subset of ATL that is verifiable.

From the point of view of the underlying *verification technique* a different possibility would have been the GROOVE tool [35]. GROOVE allows specifying, playing and analysing graph transformations. In particular GROOVE assumes that the states of the systems to be analysed are expressed as graphs and that the system's behavior is simulated by graph transformation rules that manipulate those graphs. In [36] Rensink, Schmidt and Varró test whether safety and reachability properties that are expressed as constraints over graphs can be efficiently checked by building the state space for a transformation. The answer is positive, but the authors found similar state space explosion problems as we did. In order to tackle those issues the tool relies on exploiting the symmetric nature of a problem by investigating isomorphic situations only once. This is very similar to what we do in our tool by maintaining caches throughout symbolic execution construction and property proof. Those caches allow us to avoid rerunning the expensive subgraph isomorphism algorithm as much as possible. It is foreseeable that our approach makes use of the advanced state space construction and recent CTL property checking capabilities of GROOVE. This could be achieved by using GROOVE as the transformation framework for our approach, instead of T-CORE. However, at the time of the construction of our tool, fine grained control of GROOVE transformations via an API as we do with T-CORE did not exist. It was thus unfeasible to implement our algorithms 1, 2 and 3 by relying solely on GROOVE's graphical interface.

Also from the *verification technique* viewpoint, Becker *et al.* propose a technique for checking a dynamic system which state is encoded as a graph [37]. They also use model transformations to simulate the system's progression and aim at verifying that no unsafe states are reached as part of the system's behavior. In this sense Becker *et al.*'s approach is *transformation dependent* and *input independent*, as an infinite amount of initial graphs needs to be considered. However, instead of generating the exhaustive state space as is done with GROOVE, the authors follow a different strategy by checking that no unsafe states of the system can be reached. They do so by searching for unsafe states as counterexamples of invariants encoded in the transformation rules. The

analysis is performed symbolically on the application transformation rules and as such resembles our symbolic execution technique. However, rather than being generically applicable to model transformations, possibly exogenous, the approach is geared towards the mechatronic domain and graph transformations are used as a means to encode the dynamic structural adaptation of such systems. It is thus difficult to establish a direct parallel with our work in terms of the applicability or efficiency of Becker *et al.*'s technique when applied to the verification of model syntax relations in model transformations.

9 Conclusion

In this paper we have proposed the analysis of syntactic model relation properties of model transformation via symbolic execution. We implemented our approach using model transformations written in T-Core. We have also presented early performance results. Several contributions can be identified in our work: (i) we have provided the algorithms for our original proposal of symbolic execution of model transformations in [5]. To the best of our knowledge our work provides the first attempt at explicitly building symbolic executions for a model transformation language; (ii) we show that our symbolic execution technique scales well in our experimental setting and has the potential to scale for real world problems; (iii) we demonstrate that expressiveness reduction of a model transformation language can be very beneficial to the design and construction of a model transformation verification tool; and (iv) we demonstrate that model transformations are themselves a useful tool for the proof of properties of model transformations. More generally, we provide tangible evidence that MDD principles and tools can be employed throughout the construction of MDD tools not only as mere data translators, but also at the algorithmic core of those tools.

For the future, besides exploring performance issues, we will enhance the expressiveness of our property language. Constraints on object attributes will be incorporated in the language, as will negative associations, indirect links and backward links. We are now working on applying our proof technique to model transformation properties relevant to our industrial partners, in the context of the NECSIS (Network on Engineering Complex Software Intensive Systems for Automotive Systems) project. In a different vein, it would be interesting to understand under which conditions our symbolic execution construction and proof techniques can be applied to other graph based transformation languages.

References

1. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20:42–45, Sept 2003.
2. T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.
3. Moussa Amrani, Levi Lucio, Gehan M. K. Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A tridimensional approach for studying the formal verification of model transformations. In *ICST*, pages 921–928, 2012.

4. Bruno Barroca, Levi Lucio, Vasco Amaral, Roberto Félix, and Vasco Sousa. Dsltrans: A turing incomplete transformation language. In *SLE*, pages 296–305, 2010.
5. Levi Lúcio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 136–150. Springer-Verlag, 2010.
6. Roberto Felix, Bruno Barroca, Vasco Amaral, and Vasco Sousa. Dsltrans: providing tool support for languages transformational semantics. Technical report, UNL-DI-1-2010, University Nova de Lisboa, Portugal, 2010. `http://solar.di.fct.unl.pt/twiki/pub/BATICCCS/ModelTransformationPapers/dsltrans.pdf`.
7. Cláudio Gomes and Bruno Barroca. DSLTrans User Manual. `http://msdl.cs.mcgill.ca/people/levi/files/DSLTransManual.pdf`.
8. Qin Zhang and Vasco Sousa. Practical model transformation from secured uml statechart into algebraic petri net. Technical Report TR-LASSY-11-08, `http://hera.uni.lu/~levi.lucio/verifying_access_control_statecharts/transformation_rules.pdf`, 2011.
9. David Akehurst and Stuart Kent. A relational approach to defining transformations in a metamodel. pages 243–258. Springer, 2002.
10. Anantha Narayanan, Gabor Karsai, Claudia Ermel, Reiko Heckel, Juan de Lara, Tiziana Margaria, Julia Padberg, and Gabriele Taentzer. Verifying model transformations by structural correspondence. *Electronic Communications of the EASST*, 10, 2008.
11. Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. Ocl contracts for the verification of model transformations. *ECEASST*, 24, 2009.
12. Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46, 2013.
13. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
14. Eugene Syriani and Hans Vangheluwe. De-/re-constructing model transformation languages. *ECEASST*, 29, 2010.
15. Juan de Lara and Hans Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In *FASE '02*, pages 174–188. Springer-Verlag, 2002.
16. L. Lúcio. DSLTransVerif: A Prototype Implementation, 2013. `http://msdl.cs.mcgill.ca/people/levi/files/DSLTransVerif.zip`.
17. Márk Asztalos, László Lengyel, and Tihamer Levendovszky. Towards automated, formal verification of model transformations. In *ICST*, pages 15–24, 2010.
18. Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In AGTIVE, volume 1779, pages 333–341, 2000.
19. Juan de Lara and Hans Vangheluwe. Atom^3: A tool for multi-formalism and meta-modelling. In *FASE*, pages 174–188, 2002.
20. Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In *UML*, pages 290–304, 2004.
21. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(12):31 – 39, 2008. ¡ce:title¿Special Issue on Second issue of experimental software and toolkits (EST)¡/ce:title¿.
22. Tihamer Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A systematic approach to metamodeling environments and model transformation systems in vmts. *Electr. Notes Theor. Comput. Sci.*, 127(1):65–75, 2005.
23. Detlef Plump. Termination of graph rewriting is undecidable. *Fundam. Inform.*, 33(2):201–209, 1998.

24. Juan de Lara and Hans Vangheluwe. Automating the transformation-based analysis of visual languages. *Form. Asp. Comput.*, 22(3-4):297–326, May 2010.
25. Hartmut-Karsten Ehrig, Gabriele Taentzer, Juan de Lara, Dániel Varró, and Szilvia Varró-Gyapai. Termination Criteria for Model Transformation. In FASE, 2005.
26. Dániel Varró, Szilvia Varró-Gyapai, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination Analysis of Model Transformations by Petri Nets. In ICGT, volume 4178, pages 260–274, 2006.
27. H.J. Sander. Bruggink. Towards a Systematic Method for Proving Termination of Graph Transformation Systems. ENTCS, **213**(1), 2008.
28. Jochen M. Kster. Definition and Validation of Model Transformations. SOSYM, **5**(3):233–259, 2006.
29. Detlef Plump. Confluence of Graph Transformation Revisited. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, volume 3838, 2005.
30. Reiko Heckel, Jochen M. Küster, and Gabriele Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In ICGT, 2002.
31. Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Efficient Detection of Conflicts in Graph-based Model Transformation. ENTCS, 152, 2006.
32. Enrico Biermann. Local Confluence Analysis of Consistent EMF Transformations. ECEASST, 38:68–84, 2011.
33. Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of atl transformations using transformation models and model finders. In *ICFEM*, pages 198–213, 2012.
34. Fabian Büttner, Marina Egea, and Jordi Cabot. On verifying atl transformations using 'off-the-shelf' smt solvers. In *MoDELS*, pages 432–448, 2012.
35. AmirHossein Ghamarian, Maarten Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using groove. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012.
36. Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *ICGT*, pages 226–241, 2004.
37. Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 72–81, New York, NY, USA, 2006. ACM.