- *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 'Basic Entities' layer;
- *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[2]. 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 intact as a match source. Therefore, the only possibility to reuse elements created from a previous layer is to reference 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 'StationwMale' transformation rule of layer 'Relations Layer' 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 attribute language allowing the composition of attributes of apply model elements from references to one or more match model element attributes.

3 Verifying Properties of DSLTrans Transformations

As we have mentioned in section 2, all DSLTrans transformations are, by construction, *terminating* and *confluent* [4]. We have also abstractly shown in [23] how to build a symbolic execution such that properties of DSLTrans transformations can be proved. We formally prove in [23], proposition 2, that such a symbolic execution is always finite. With our current work we aim at producing a tool that can prove or disprove such properties efficiently.

[2] In the implementation the notion of indirect links only captures EMF containment associations in order to avoid cycles.

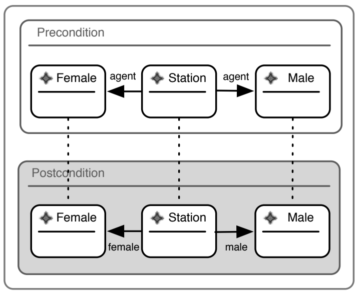

Fig. 4: Police Station Transformation Property 1

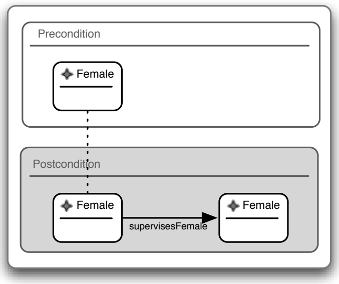

Fig. 5: Police Station Transformation Property 2

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.
The properties we are interested in proving are thus 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 impose traceability relations between precondition and postcondition elements. A formal definition of our property language can be found in [23].
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

The construction of our symbolic execution begins by building the powerset of all the rules in the first layer of the transformation (layer *Entities*). This means that we are building all the possible combinations of applications of rules in the first layer. Each such rule combination represents a symbolic execution of the first layer and we will henceforth refer to any combination of rules of a DSLTrans transformation as a *state*.

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 [19] on the topic. According to King, by performing symbolic execution of a program we can produce a set of conditions on that program's input variables. Each of those conditions represents a possible path through the program, but applies to many (possibly an infinite amount of) values for those input variables. In that sense each condition representing a path through the program is symbolic. In our case, each rule application we assume during the state space construction poses conditions on the input (but also output) models. In particular for our example, if we say that only an element of type 'Station' can be consumed, then we are assuming that our model only contains *stations* (or possibly other elements in the metamodel not tackled by the rules in the current layer being treated). However, even if we assume *stations* are being consumed by rule *Station2Station* (see figure 2), we may not know how many *stations* we have in our input model. This abstraction over the number of times the rule has matched is what makes it such that a state of our symbolic execution is indeed 'symbolic'.

It is also important to note at this point that the fact that all DSLTrans' transformations are confluent allows us to disregard the order of the application of the rules within each symbolic state. In fact the confluence proof for DSLTrans' transformations presented in [4] is based on the fact that the result of applying the rules within a layer in any given order to an input model is deterministic. If this would not be the case then the ordering of the rules within a layer would have to be considered during symbolic execution, which would lead to a multiplication of the considered symbolic states and thus to an even more pronounced state space explosion.

After having processed the first layer in the construction our symbolic execution for the transformation in figure 2, we can now proceed to the second layer. As for the first layer, the powerset of all the rules in the second layer is calculated. We now need to understand how each one of these newly built states affects the partial symbolic execution built previously from the first layer. When we analyse a state belonging to the powerset of the second layer (noted $State_{l2}$) against a state belonging to the symbolic execution built so far by the rules in the first layer (noted $State_{l1}$), several cases may occur:

1. If none of the rules in the $State_{l2}$ contains backward links, a new state is added to the symbolic execution by extending $State_{l1}$ by uniting $State_{l1}$ and $State_{l2}$. This union is built adding the rules in $State_{l2}$ to the rules in $State_{l1}$;
2. if the rules in $State_{l2}$ includes backward links, then we need to analyse whether those backward links correspond to traces between match and apply elements generated by rules in $State_{l1}$. If this is not the case then the conditions for at least one of the rules from $State_{l2}$ to apply is not satisfied and $State_{l2}$ cannot be added to the symbolic execution. Figure 6 illustrates this case, given that the backward link the Station2Female rule in $State_{l2}$ connecting the match element *Female* to the apply element *Female* does not have a corresponding trace[3] in $State_{l1}$;

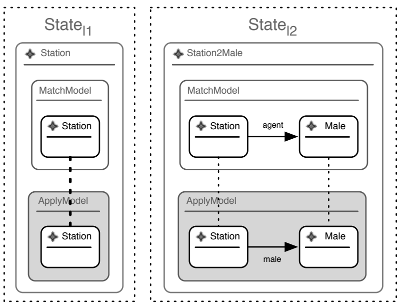

Fig. 6: Non Mergeable Symbolic States

3. if the rules in $State_{l2}$ include backward links and all those backward links correspond to traces between match and apply elements generated by rules in $State_{l1}$ then, as in point 1, $State_{l1}$ can be extended by $State_{l2}$. An example of this case can be observed in figure 7. However, in this case the extension is performed slightly differently: all rules from $State_{l2}$ containing backward links are merged with the rules from $State_{l2}$ where the traces corresponding to the backward links were generated. Additionally, $State_{l1}$ is removed from the symbolic execution. This is because, given we assume all elements necessary for rules of $State_{l2}$ including backward links were generated by the rules of $State_{l1}$ the rules from $State_{l2}$ with backward links necessarily execute. As such, $State_{l1}$ can no longer exist on its own in the symbolic state space;
4. a slight variation of case 3 is the case where more than one backward link from the same rule in $State_{l2}$ is matched over the same trace of a rule from $State_{l1}$. An example of this case can be observed in figure 8. In this case, additionally to what happens in case 3, $State_{l1}$ also needs to be kept in the symbolic execution. This is due to our abstraction over the number of

[3] 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.

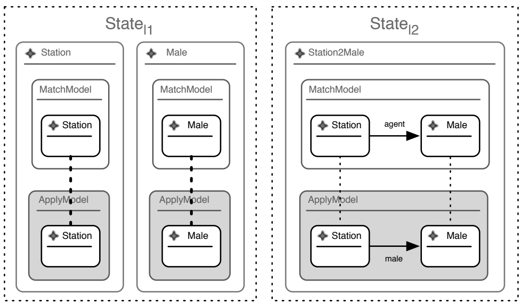

Fig. 7: Mergeable Symbolic States

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 can apply or not. We thus need to consider both cases.

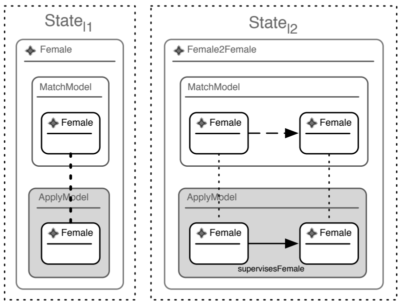

Fig. 8: Mergeable Symbolic States with Repetition

The informally introduced symbolic execution construction technique is detailed in algorithm 1. We introduce in the algorithm some predicates such as e.g. *exist-BackwardLinks* or *noOvelappingMatchesExist*, for which no further description is provided but which semantics should be deducible from the introduction above.

3.2 Property Proof

We need to check for each state of the completed symbolic execution whether, if the precondiction holds of the property holds, then the postcondition of the property also holds. The fact that the precondition holds for a state can be directly checked by matching the property graph on the union of all the rules in the final state. However, as we have partially described in [23], because in the general

Algorithm 1 Symbolic Execution Generation

```
1: procedure GENSTATESPACE($transf$)
2:     $symbStateSpace = \emptyset$
3:     for $curLayer \in transf$ do
4:         if curLayer = firstLayer then
5:             $symbStateSpace = \mathcal{P}(curLayer)$
6:         else
7:             $curLayerStateSpace = \mathcal{P}(curLayer)$
8:             $statesToAdd = \emptyset$
9:             for $State_{l+1} \in curLayerStateSpace$ do
10:                if existBackwardLinks($State_{l+1}$) then
11:                    for $State_l \in symbStateSpace$ do
12:                        if $\forall backLink \in State_{l+1}, \exists rule \in State_l . backLink \in traces(rule)$
                               then
13:                            $mergedState = mergeRulesOverBackwardLinks(State_l, State_{l+1})$
14:                            $statesToAdd = statesToAdd \cup mergedState$
15:                            if noOvelappingMatchesExist(mergedState) then
16:                                $symbStateSpace = symbStateSpace \setminus \{State_l\}$
17:                            end if
18:                        end if
19:                    end for
20:                else
21:                    $statesToAdd = statesToAdd \cup (State_l \cup State_{l+1})$
22:                end if
23:            end for
24:        end if
25:        $symbStateSpace = symbStateSpace \cup statesToAdd$
26:    end for
27: end procedure
```

case we do not know whether two match elements of the same type occuring in two different rules in the same final symbolic state consume the same instance of that type or not in a concrete input model, we need to consider two cases: (1) the case where those two distinct match elements from different rules consume two different instances of that type in the input model; and (2) the case where those two match elements consume the same instance in the input model. This is achieved by building for each final state of the symbolic execution all the possibilities of merges of elements of the same type which belong to different rules. This is achieved by algorithm 2, called the *Collapse* algorithm.
We will exemplify the *collapse* algorithm on the state in figure 9. The algorithm starts by finding all the pairs of elements of the same type in the state's rules. 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. Figure 10 shows the result of applying this step to our example. Note that the choice of the pair's match element that receives all links belonging to both elements is non deterministic. Note also that the two original

Algorithm 2 Collapse

```
 1: procedure COLLAPSE($state$)
 2:     $collapsableMatchPairSet = getSameTypeDiffRulesMatchPairs(state)$
 3:     $collapsedStateSet = \emptyset$
 4:     for $matchPair \in collapsablePairSet$ do
 5:         $state' = moveAllLinks(matchPair, state)$
 6:         if $matchPair = (m_1, m_2) \wedge \exists(a_1, a_2).backwardlink(m_1, a_1) \wedge$
                $backwardlink(m_2, a_2) \wedge sameType(a_1, a_2)$ then
 7:             $state' = moveAllLinks((a_1, a_2), state')$
 8:         end if
 9:         $stateSet = stateSet \cup state' \cup Collapse(state')$
10:     end for
11:     return stateSet
12: end procedure
```

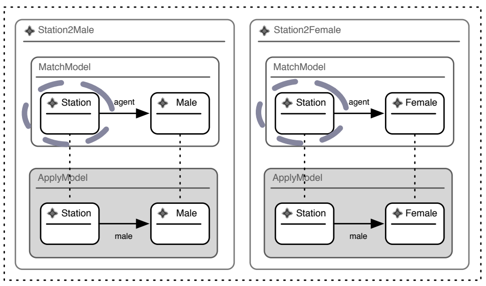

Fig. 9: Collapse algorithm: locating two Match elements of the same type

rules now become one, with the name of the new rule being the concatenation of the two original rules' names.
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, then those apply elements need to be merged in the same way as the match ones were. 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 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.
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 2 will then proceed by adding to the set of collapsed states to return: (1) the result of the current collapse step; (2) the recursive result of collapsing the rules remaining from the current collapse step. The algorithm will then loop over the

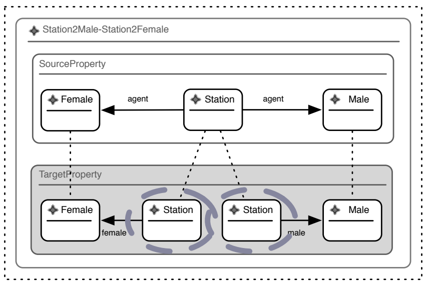

Fig. 10: Collapse algorithm: merging two Match elements and locating two Apply elements of the same type

remaining pairs of match elements having the same type until no more pairs exist and will return all produced collapsed states.

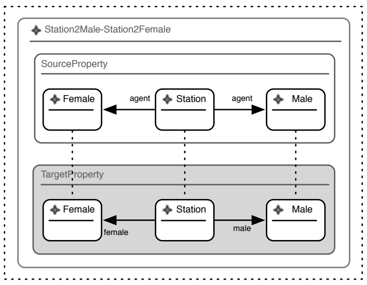

Fig. 11: Collapse algorithm: merging two Apply elements

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. Since there are dependencies between the steps, in what follows we list the steps in the order they must occur. 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

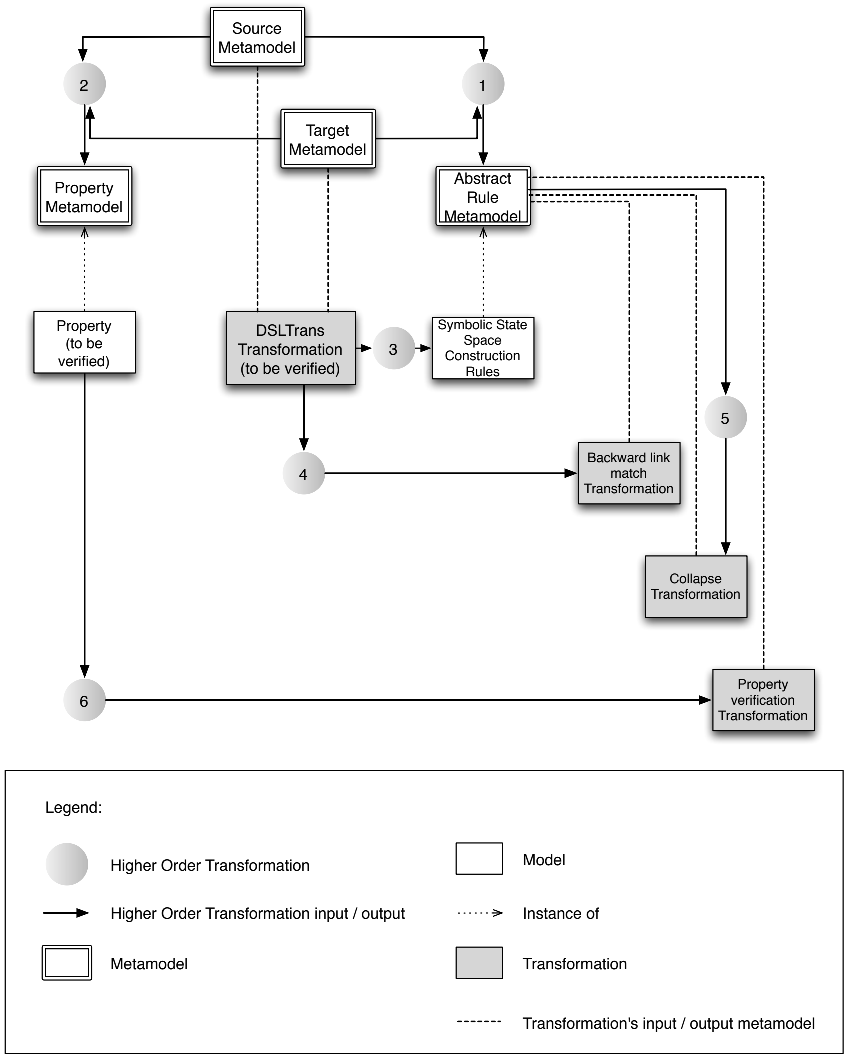
Source Metamodel
2
1
Target Metamodel
Property Metamodel
Abstract Rule Metamodel
Property (to be verified)
DSLTrans Transformation (to be verified)
3
Symbolic State Space Construction Rules
5
4
Backward link match Transformation
Collapse Transformation
6
Property verification Transformation
Legend:
Higher Order Transformation
Model
Higher Order Transformation input / output
Instance of
Metamodel
Transformation
Transformation's input / output metamodel

Fig. 12: Verification Tool Architecture

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 higher order transformations numbered 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 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 transformation under analysis (TUA). It takes as input the source and target metamodels for the TUA and returns the language in which properties are written;
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*;
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*;
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*;
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 these query transformations have as source metamodel the *abstract rule metamodel*.

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. What better tool to use for such a task than a model transformation language? As described in chapter 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. These model transformations are independent of the TUA and are part of

our toolset. Then, model transformations generated during the first step can be used 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 verification algorithms 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 [30]. 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 Scalability 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 [9] 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* 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 2;
– 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. Given T-Core is built as a Python library, T-Core primitives can be embedded in 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 and 2.
Whenever possible we have taken advantage of the fact that the elementary rules and the computations required by the symbolic execution and collapse algorithms (algorithms 1 and 2) are used repetitively in order to save in memory and computation time. In what concerns algorithm 1, we have used pointers to rules instead of copies of rules to build each state. Caches were used to accelerate the repetitive backward link match transformations and the *mergeRulesOverBackwardLinks* operations between rules of different layers. For algorithm 2, instead of the recursive strategy explained in section 3 we have implemented a method to build the set of collapsed states for a given state incrementally. The strategy involved starting by collapsing rules in a state two by two, then collapsing again the results with remaining rules, and so on. This solution works because the collapse operation is commutative. In this fashion no repeated collapsed states are built and intermediate results of collapsing several rules can be cached to be used when collapsing the same rules in different states. It may however be the case that, if no or few

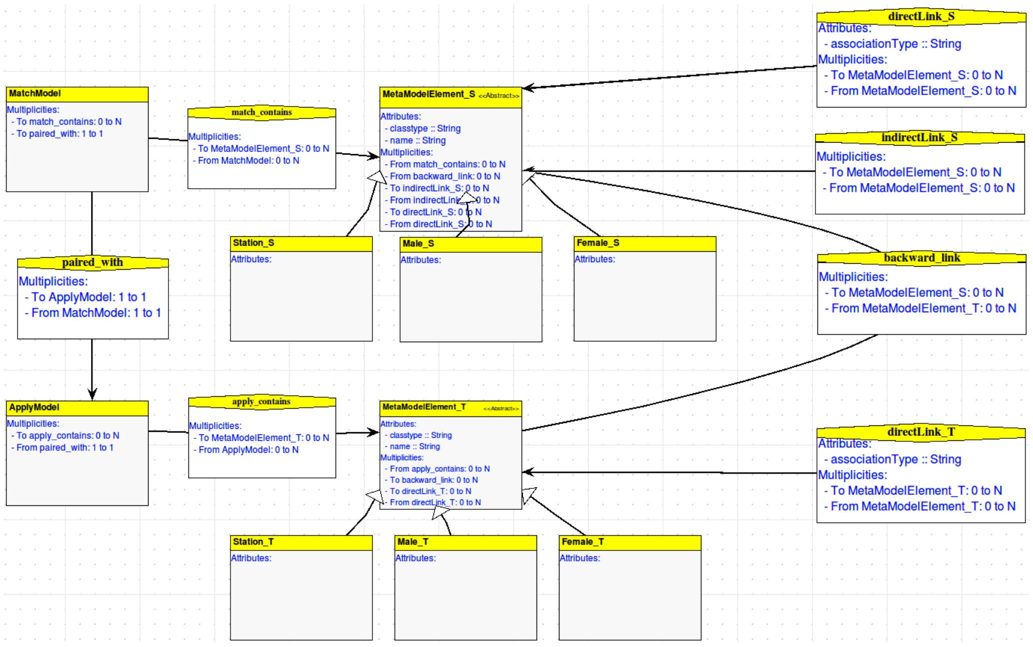
MatchModel
Multiplicities:
- To match_contains: 0 to N
- To paired_with: 1 to 1
match_contains
Multiplicities:
- To MetaModelElement_S: 0 to N
- From MatchModel: 0 to N
MetaModelElement_S <<Abstract>>
Attributes:
- classtype :: String
- name :: String
Multiplicities:
- From match_contains: 0 to N
- From backward_link: 0 to N
- To indirectLink_S: 0 to N
- From indirectLink_S: 0 to N
- To directLink_S: 0 to N
- From directLink_S: 0 to N
directLink_S
Attributes:
- associationType :: String
Multiplicities:
- To MetaModelElement_S: 0 to N
- From MetaModelElement_S: 0 to N
indirectLink_S
Multiplicities:
- To MetaModelElement_S: 0 to N
- From MetaModelElement_S: 0 to N
paired_with
Multiplicities:
- To ApplyModel: 1 to 1
- From MatchModel: 1 to 1
Station_S
Attributes:
Male_S
Attributes:
Female_S
Attributes:
backward_link
Multiplicities:
- To MetaModelElement_S: 0 to N
- From MetaModelElement_T: 0 to N
ApplyModel
Multiplicities:
- To apply_contains: 0 to N
- From paired_with: 1 to 1
apply_contains
Multiplicities:
- To MetaModelElement_T: 0 to N
- From ApplyModel: 0 to N
MetaModelElement_T <<Abstract>>
Attributes:
- classtype :: String
- name :: String
Multiplicities:
- From apply_contains: 0 to N
- To backward_link: 0 to N
- To directLink_T: 0 to N
- From directLink_T: 0 to N
directLink_T
Attributes:
- associationType :: String
Multiplicities:
- To MetaModelElement_T: 0 to N
- From MetaModelElement_T: 0 to N
Station_T
Attributes:
Male_T
Attributes:
Female_T
Attributes:

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

collapse operations are required for a state, 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 2 directly.

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

In order to understand the scalability 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 scalability 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 two times, in order to reach a maximum of 21 rules distributed by 4 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

# of rules	5	7	12	14	19	21	28
# of states	14	31	337	1051	11428	35641	1208641
symbolic execution construction time (sec)	0.12	0.25	0.40	0.93	9.88	53.27	30513.64
used memory (Kb)	93×10^{-3}	0.17	1.41	4.40	48.16	139.35	-
Prop. 1, repl. 1 (sec)	0.11	0.68	2.21	6.97	88.57	320.00	-
Prop. 1, repl. 2 (sec)	-	-	1.41	7.06	89.87	347.01	-
Prop. 1, repl. 3 (sec)	-	-	-	-	78.92	323.50	-
Prop. 2, repl. 1 (sec)	1.8×10^{-3}	1.8×10^{-3}	1.7×10^{-3}	1.6×10^{-3}	1.6×10^{-3}	1.6×10^{-3}	-
Prop. 2, repl. 2 (sec)	-	-	0.04	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	-

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

correspond to different source and target metamodel elements. As an example, the *S1* match element matches different model elements than match element *S2*.

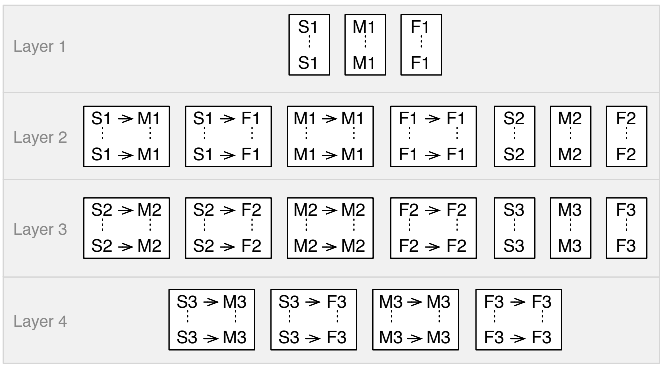

Fig. 14: Replicated Police Station Transformation for Scalability Tests

In table 1 we present some scalability 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. For each measurement involving time we repeated the given experiment thrice and calculated the final result as the average of the three experiment results.

The first line of the table 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.
The second and third lines in table 1 present respectively the number of states and the computation time for the symbolic execution construction for the given amount of rules. Both the number of states 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 for 21 rules. In order to test the limits of our approach we have tried building the symbolic execution of 28 rules, by adding a fourth replica of the Police Station transformation. In this case the number of built states is over 1 million and the time to do so over 8 hours. We have not attempted to prove properties for such a large symbolic execution.
Lines 4 through 6 of table 1 present the times to prove property 4 of the Police Station transformation. 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 states 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 7 through 9 of table 1 present the times to disprove property 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 symbolic states involving those rules later and requires more time.
Finally line 10 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.

7 Discussion of the Results and Contributions

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

- The algorithms to build the space state for a given DSLTrans' model transformation and prove structural properties of that transformation based on the constructed state space. This work materialises our proposal originally presented in [23];

– A first implementation of those algorithms based on the T-Core model transformation framework [30] is described. Based on our running example we have performed a detailed scalability 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 in the context of our experiments;
– The architecture of tool to allow for performing the analysis of any DSLTrans transformation is given. We have produced by hand the artifacts necessary for building the state space 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;
– The proof that a symbolic state space can be practically built 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 state space 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 scalability results we have presented in this paper are promising, but further experiments with other transformations need to be done in order to access the real scalability 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 languages, inspiration can be drawn from several proposals in the literature [25, 8, 3, 16], 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 state space construction such conditions will have to be addressed symbolically, which adds an additional challenge to DSLTrans' symbolic execution construction.

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 [2]. 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 [31], AToM^3 [10], VIATRA2 [32], ATL [18] 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 [26], termination criteria and techniques for analysing such criteria on transformations written in graph based transformation languages [11, 12, 33, 7, 20] have been proposed to alleviate this problem. Confluence is also undecidable for graph based transformation languages [27]. As for termination, several confluence criteria and corresponding analysis techniques have been proposed in the literature [17, 20, 21, 6].

Regarding the *properties* of transformations that can be checked, according to the classification in [2] 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 [1]. 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 [25]. 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