A Technique for Automatic Validation of Model Transformations

Levi Lúcio, Bruno Barroca, Vasco Amaral

Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal**
{Levi.Lucio,Bruno.Barroca,Vasco.Amaral}@di.fct.unl.pt

Abstract. We present in this paper a technique for proving properties about model transformations. The properties we are concerned about relate the structure of an input model with the structure of the transformed model. The main highlight of our approach is that we are able to prove the properties for all models, i.e. the transformation designer may be certain about the structural soundness of the results of his/her transformations. In order to achieve this we have designed and experimented with a transformation model checker, which builds what we call a state space for a transformation. That state space is then used as in classical model checking to prove the property or, in case the property does not hold to produce a counterexample. If the property holds this information can be used as a certification for the transformation, otherwise the counterexample can be used as debug information during the transformation design process.

1 Introduction

Nowadays model transformation tools [9, 11] have become the topic of intensive research due to their importance in Model Driven Development (MDD). In the MDD context, these tools are used for several activities such as model refinement, refactoring, translation, validation or operational semantics. These activities can turn out to complex and error prone. This said, automatic validation techniques for model transformations are of the utmost importance.

In our laboratory, we have developed a new tool named DSLTrans [3] to assist the software engineer while specifying model transformations. DSLTrans aims at overcoming the flaws of state of the art model transformation tools — most importantly lack of confluence and termination guarantees — by proposing a simple visual language with basic primitives. The main idea behind DSLTrans is that, due to its simplicity, we can assure these features by construction.

** The presented work has been developed in the context of project BATIC3S partially funded by the Portuguese foundation FCT/MCTES ref. PTDC/EIA/65798/2006, the doctoral grant ref. SFRH/BD/38123/2007 and the post doctoral grant ref. SFRH/BPD/65394/2009. We would also like to thank Vasco Sousa and Carla Ferreira for the fruitful discussions.

In this paper, we present a technique for automatic validation of model transformations expressed in DSLTrans. We will describe a symbolic model checker which was built to guarantee transformation properties expressed in the form of an implication: ‘if a structural relation between some elements of the source model holds, then another structural relation between some elements of the target model should also hold’. Our symbolic model checker computes for each possible execution of a transformation, an equivalence class representing a set of source models and their corresponding transformations. We can then validate that transformation by checking if our transformation property holds for every computed equivalence class.

1.1 Related Work

In order to aid the construction of the proof of semantic preservation along a set of transformation rules [2] introduced a language to anotate those rules with assertions. The idea is to then pass these annotations to a reasoning framework that will derive, at the meta level, conclusions about the overall transformation. The work presented in [1] aims at validating a model transformation by using the Alloy tool. In this case, Alloy simulates the transformation by generating a model example of the source language and then analyzing the results of the transformation.

The authors of [5] present a constructive fashion to automatically generate a valid transformation (the authors refers to transformations as ontology alignment) which in principle would preserve the semantic properties of the input and output models. This generation is done by using the Similarity Flooding algorithm which is based on the calculus of a distance measurement between source and target languages.

Similarly to our approach, the authors of [10] enable the declaration of a syntactic structural correspondence between terms in source and target languages. However, they use this structural correspondence to automatically verify the results at the end of each transformation. With this approach, the quality engineer will only realize that the transformation is invalid when some pair of models input/output violates the declared structural correspondence.

1.2 Structure of the paper

This paper is organized as follows. In section 2, we introduce the DSLTrans language by providing a transformation which we use as running example throughout this paper. We then present how the state space is built for the transformation and how that state space is used to prove some properties; In section 3 we introduce the formalization of our approach with the aim of having a precise description of the transformation model checker and a base for its implementation; In section 4, we will describe how we have implemented the transformation model checker using SWI-Prolog and provide a notion of the space complexity of our algorithm; finally, in section 5, we finish with some technical directions on how to improve the space and time complexities of our transformation model checker.

2 Motivating Example

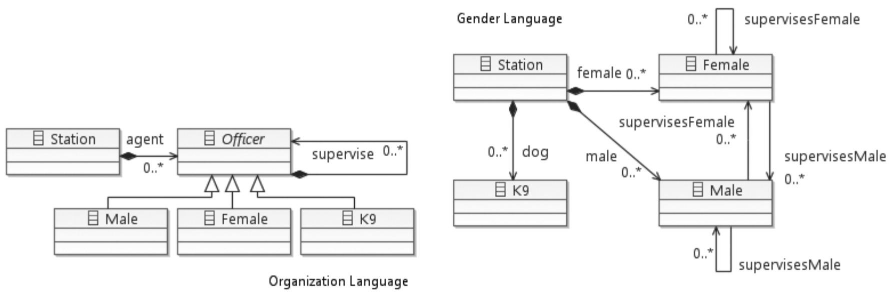

Fig. 1. Metamodels of a squad of agents(left) and a squad organized by gender(right).

2.1 The transformation language

The transformation language we use as a base for this work is called DSLTrans [3] and was developed in our laboratory. DSLTrans is a relatively simple transformation language having a reduced number of primitives. In order to introduce DSLTrans let us first present the running example we will use throughout this paper. Fig. 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 includes male officers ($Male$ class), female officers ($Female$ class), and dogs ($K9$ class). The officer chain of command is expressed using the EMF's containment named 'supervise'. 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 Fig. 2 we present a transformation written in DSLTrans between models of both languages. The purpose of this transformation is to flatten a chain of command given in language 'Organization Language' into two 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. An example of one such transformation can be observed in Fig. 3, where the original model is on the left and the transformed one on the right. Notice that in the figure the boxes represent instances of the classes in the metamodels of Fig. 1. In particular, the elements s, m_k and f_k in the figure

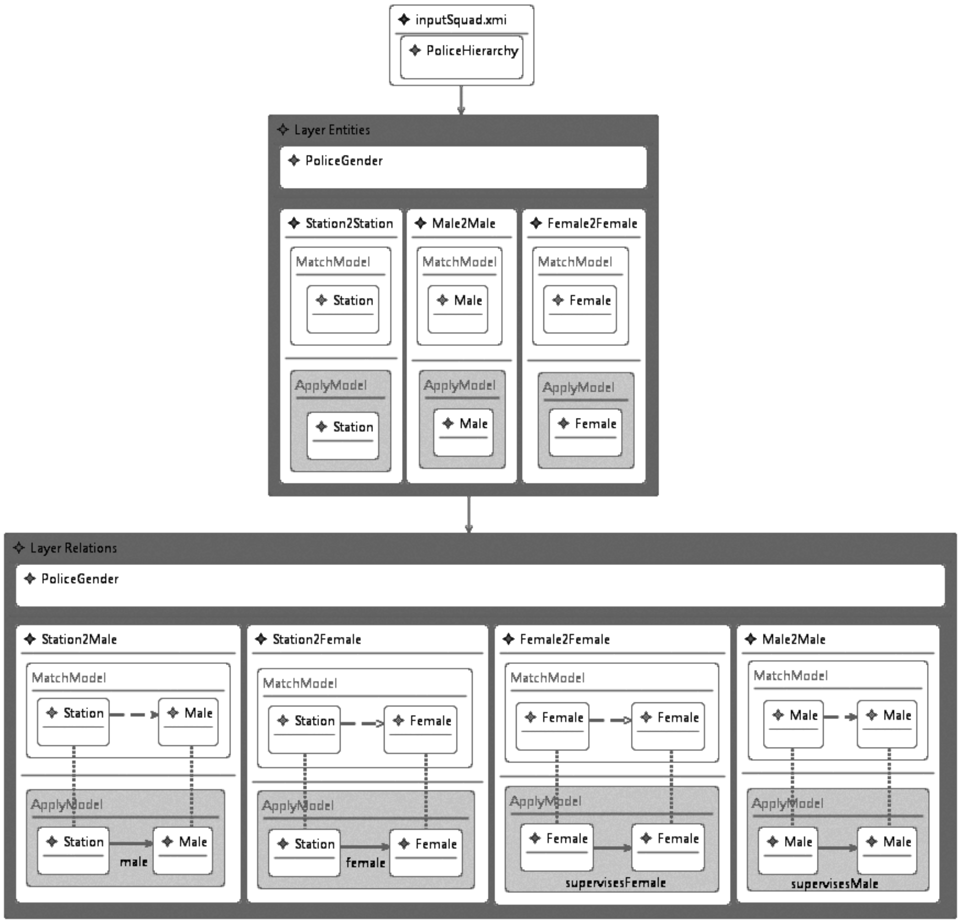

Fig. 2. A model transformation expressed in DSLTrans.

on the left are instances of the source metamodel elements *Station*, *Male* and *Female* respectively. The primed elements in the figure on the right are their instance counterparts in the target metamodel.

We can identify in the transformation in Fig. 2 several components. Firstly, the transformation is divided into two steps, formally called *layers*. Each layer defines a set of transformation rules and a transformation rule is a pair $\langle match, apply \rangle$, where both *match* and *apply* are patterns holding elements of the metamodel of language 'Squad Organization Language' — the source language — and of language 'Squad Gender Language' — the target language — respectively. Layer 1 (named 'Layer Entities') of the transformation includes three simple transformation rules to translate elements of a model of language 'Organization Language' into their counterparts in language 'Gender Language'. Layer 2 (named 'Layer

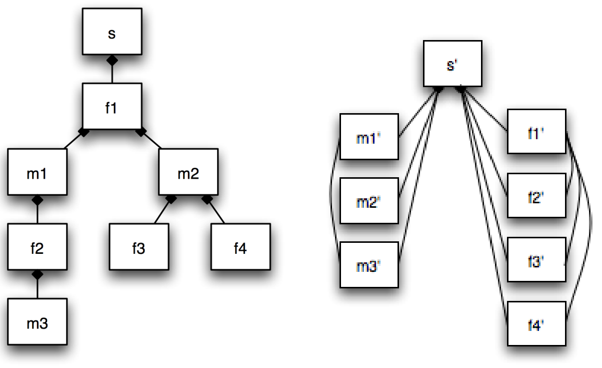

Fig. 3. Original model (left) and transformed model (right).

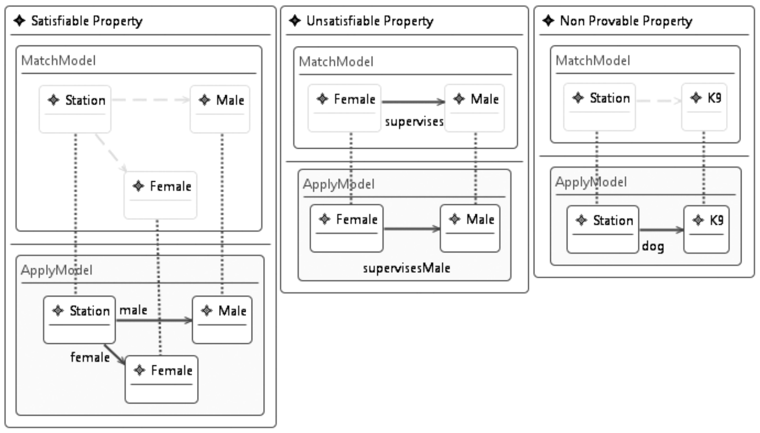

Fig. 4. Validation properties over a DSLTrans model.

Relations') includes four transformations that give structure to the elements built in the previous layer. The transformation rules in layer 2 reveal two interesting features of DSLTrans:

- *Indirect links*: these links can be observed in the *match* pattern of all the transformations of layer 2 and are noted as a dashed arrow. A model matches such an indirect link if there exists a path of containment associations between instances of the two connected metamodel elements;
- *Backward links*: backward links connect elements of the *match* and the *apply* patterns and are noted as dashed (vertical) lines. They can also be observed in all the transformation rules of layer 2. Backward links are used to refer to elements created in a previous layer in order to use them in the current one. For example, the leftmost transformation rule of layer 2 in Fig. 2 takes

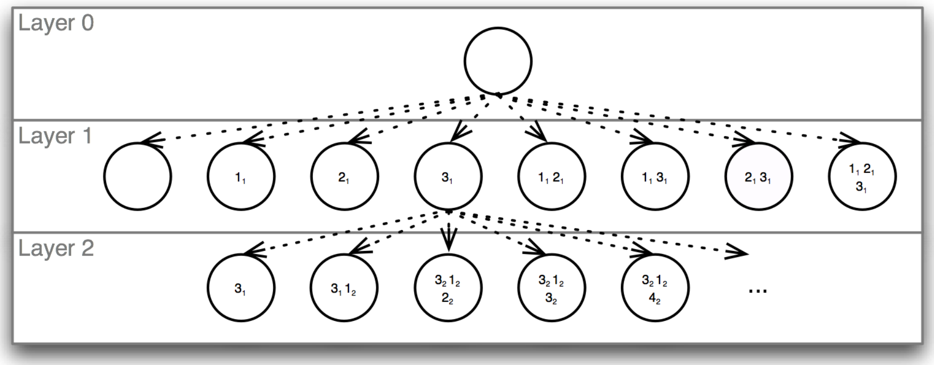

Fig. 5. Partial state space for the transformation in Fig. 2.

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 creates an association 'male' between them.

A particular characteristic of DSLTrans as a transformation language is that throughout all the layers the source model remains intact as a match source. The *match* pattern of a transformation rule can match multiple times the source model and per each of those matches an instance of the *apply* pattern is created. Each layer thus creates a set of target metamodel instances and relations between those instances. In order to refer to elements created in a previous layer in a transformation rule, backward links have to be used. A complete description of the DSLTrans language including its formal syntax and semantics can be found in [3]. An example of a complex transformation of UML to Java using DSLTrans can be found in [8].

2.2 Properties and their proof

Now that the transformation language has been defined we can move on to describe the properties we wish to prove about our transformations. Examples of these properties can be observed in Fig. 4. In natural language the property named 'Satisfiable Property' reads as follows: 'Any 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'. The primary goal of our model checker is to prove that, given a transformation, such a property will hold for all models given as inputs to that transformation.

Practically, this proof is achieved by building what we call the state space of a transformation. Each state of the transformation state space corresponds to a possible combination of the transformation rules of a given layer, combined with all states of the previous layer. Using the example of the transformation given in Fig. 2 we can build a rough sketch of such a state space which we present in Fig.5. In the figure we identify each transformation rule in each layer by a number with an index. For example transformation 1_1 corresponds to the first transformation — e.g. left to right in Fig. 2 — in layer one. The state space starts with the initial state— which in the figure belongs to layer 0 — where no transformation has been applied. The initial state then connects to all possibilities of combinations of transformation rules in layer 1. Each of the states produced by layer 1 is then connected to all possibilities of combinations of transformation rules in layer 2 — in the figure we only exemplify with the state 3_1. The states in layer 2 include not only the combinations of transformation rules from that layer, but also the transformation rules coming from a state produced by the previous layer. In such a way each state accumulates all transformation rules leading to it and thus a describes pattern(s) that should exist in the source model. As such, each state symbolically describes an equivalence class of input models.

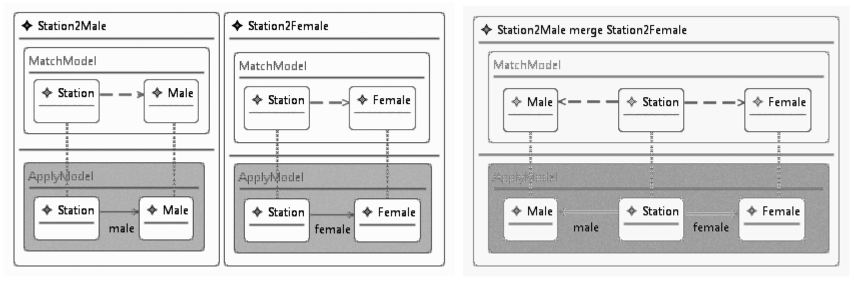

Fig. 6. Original transformations rules (left) and a possible collapse of those rules (right).

We can be more precise while building such equivalence classes. In Fig. 6 we exemplify what we call the *collapse* of two of the rules of the transformation in Fig. 2. Due to the semantics of DSLTrans it may occur that, for example, if we have the two transformations on the left of Fig. 6 applied to a model, the instance of *Station* used by the match pattern of the two rules is the same. This comes from the fact that, in DSLTrans, the same input can consumed by several transformation rules within the same layer. In this case we can collapse the two classes in one in the state we are building. In fact, we can even go further and collapse the *Station* classes in the apply pattern of the two rules which would mean that both *Station* instances previously created in layer 1 (notice the backward link) are actually the same. This leads to the state shown in Fig. 6 on the right. In fact this state is required to prove the 'Satisfiable Property' in Fig. 4.

More generally, *collapsing* transformation rules is used to add more definition to the equivalence classes represented by each state than the simple union of transformations as can be seen in Fig. 2. In this union, all elements of the same type in the disjoint graphs of the united transformation are seen as referring to different objects in the input model — i.e. several elements of the same type within a transformation necessarily refer to different objects in a model. By adding the collapsed transformation rule states to the state space, the proofs of our properties become complete given we are covering more models in our symbolic states.

The proof of a property is then achieved by walking through the state space and checking every complete transformation state space path (starting from the initial state): if there is a state that satisfies the *match* pattern of the property, then there must exist a subsequent state for which the *apply* pattern satisfies the *apply* pattern of the property.

In Fig.4, the property named 'Unsatisfiable Property' represents a property that is not true for the transformation in Fig. 2. In natural language the property states the following: 'If a male officer commands a female officer in the original model, then that relation will be preserved in the transformed mode'. In our simple example, this is clearly not true given that the point of our transformation is to build separate lists of male and female officers. That said, in order to be proved, the property should hold on all paths of the state space, therefore it is sufficient to find one path where the property does not hold to render the property false. Such a path can then be used as a counterexample and may be useful for the transformation designer in the sense that it may point out a sequence of transformation rules leading to a wrong transformation result.

It may also happen that a property is non provable. In Fig. 4, the property named 'Non Provable' refers to dogs in the *match* pattern, a situation which is never contemplated by the transformation rules in Fig. 2. As such, the only possible statement about this property is that, although the source metamodel would allow such *match* patterns, the transformation does not implement them. This situation may point out to the transformation designer that (s)he is missing transformation rules to address certain patterns of the input models.

3 Formalization

In this section we will present the detailed theory for our transformation symbolic model checker. The theory is introduced incrementally and it formalizes the informal description given in section 2. The goal of such a formalization is to provide a precise definition of our symbolic model checker, to abstractly build the algorithms to perform the proofs and to provide a base for the study of the complexity of such algorithms. The formalization we provide tackles the core syntax and semantics of our symbolic model checker, but for tractability reasons leaves out: negative conditions in transformation rules; dealing with class attributes; inheritance and other complex relations in metamodels and their instances. Moreover, the proofs for the propositions stated during the formalization can be found at [6].

3.1 Graph definitions

Definition 1. *Typed Graph*

A typed graph is a triple $\langle V, E, \tau \rangle$ where V is a finite set of vertices, $E \subseteq V \times V$ is a set of edges connecting the vertices and $\tau : V \rightarrow Type$ is a typing function for the elements of V, where $Type$ is a set of type names. Edges $(v, v') \in E$ are noted $v \rightarrow v'$. The set of all typed graphs is called TG.

Definition 2. *Typed Graph Union*

Let $\langle V, E, \tau \rangle, \langle V', E', \tau' \rangle \in TG$ be typed graphs. The typed graph union is the function $\sqcup : TG \times TG \rightarrow TG$ defined as:

$$\langle V, E, \tau \rangle \ \sqcup \ \langle V', E', \tau' \rangle = \langle V \cup V', E \cup E', \tau \cup \tau' \rangle$$

Definition 3. *Typed Subgraph*

Let $\langle V, E, \tau \rangle = g, \langle V', E', \tau' \rangle = g' \in TG$ be typed graphs. g' is a typed subgraph of g, written $g' \blacktriangleleft g$ iff for all $v'_1 \rightarrow v'_2 \in E'$ there is a $v_1 \rightarrow v_2 \in E$ such that $\tau'(v'_1) = \tau(v_1)$ and $\tau'(v'_2) = \tau(v_2)$.

Notice that the notion of subgraph in the context of typed graphs is not directly concerned with the topology of the involved graphs, but rather with the topology of the nodes having the same type.

3.2 Metamodel, Model and Transformation definitions

We start by defining the notion of metamodel. A couple of metamodels were introduced in Fig. 1 and can be seen as typed graphs where the nodes are classes and the edges are associations.

Definition 4. *Metamodel*

A metamodel $\langle V, E, \tau \rangle \in TG$ is a typed graph where τ is a bijective typing function. The set of all metamodels is called $META$.

Formally, a metamodel corresponds to a graph of typed elements where only one element for each type is represented.

Let us now define the notion of model. Two models can be observed in Fig. 3 and can also be seen as typed graphs, instances of a given metamodel. Only, as can be observed in Fig. 3, models can have several instances of the same type.

Definition 5. *Model*

A model is a 4-tuple $\langle V, E, \tau, M \rangle$ where $\langle V, E, \tau \rangle$ is a typed graph. Moreover $M = \langle V', E', \tau' \rangle \in META$ is a Metamodel and the codomain of τ equals the codomain of τ'. Finally $\langle V, E, \tau \rangle \blacktriangleleft M$, which means $\langle V, E, \tau \rangle$ is an instance of a metamodel M. The set of all models for a metamodel M is called $MODEL^M$.

Definition 6. *Match-Apply Model*

A Match-Apply Model is a 6-tuple $\langle V, E, \tau, Match, Apply, Bl \rangle$, where $Match = \langle V', E', \tau', s \rangle$ and $Apply = \langle V'', E'', \tau'', t \rangle$ are models, $V = V' \cup V''$, $E =$

$E' \cup E'' \cup Bl$ and $\tau = \tau' \cup \tau''$. Edges $Bl \subseteq V' \times V''$ are called backward links. *s is called the* source *metamodel and t the target metamodel. The set of all Match-Apply models for a source metamodel s and a target metamodel t is called MAM_t^s.*

A match-apply model is an extended definition of a model which is suited to define the semantics of a model transformation. Given the semantics of DSLTrans which keeps the source model unchanged and modifies the apply model as several transformations are applied, a match-apply model is a suitable formalism to store the intermediate steps of a transformation. In particular, the *backward links* allow keeping a history of which elements in the match model created which elements in the apply model.

Definition 7. *Transformation Rule*
A Transformation Rule is a 7-tuple $\langle V, E \cup Il, \tau, Match, Apply, Bl, Il\rangle$, where $\langle V, E, \tau, Match, Apply, Bl\rangle \in MAM_t^s$ is a match-apply model. $Match = \langle V', E', \tau', s\rangle$ and the edges $Il \subseteq V' \times V'$ are called indirect links. *The set of all transformation rules having source metamodel s and target metamodel t is called TR_t^s.*

We define a *transformation rule* as a particular kind of match-apply model which allows *indirect links* in the *match* pattern, but not in the *apply* one. The reason for this is that *match* patterns can be more abstract than models, but *apply* patterns define — in fact build — instances of models. In Fig. 2, we have presented several examples of transformation rules.

Definition 8. *Property*
A Property is a 7-tuple $\langle V, E \cup Il, \tau, Match, Apply, Bl, Il\rangle$, where $\langle V, E, \tau, Match, Apply, Bl\rangle \in MAM_t^s$ is a match-apply model. $Match = \langle V', E', \tau', s\rangle$, $Apply = \langle V'', E'', \tau'', t\rangle$ and the edges $Il \subseteq (V' \times V') \cup (V'' \times V'')$ are called indirect links. *The set of all properties having source metamodel s and target metamodel t is called $Property_t^s$.*

The language to describe properties is in fact very similar to the language to express transformations, with the additional possibility of expressing indirect links in the *apply* pattern — thus allowing more abstract patterns than the ones expressed in transformations. This is natural given that the properties of a transformation can be more abstract than the rules implementing them.

Finally, we define *layers* as sets of transformation rules and *transformations* as lists of layers.

Definition 9. *Layer, Transformation*
A layer is a finite set of transformation rules $tr \subseteq TR_t^s$. The set of all layers for a source metamodel s and a target metamodel t is called $Layer_t^s$. A transformation is a finite list of layers denoted $[l_1 :: l_2 :: \ldots :: l_n]$ where $l_k \in Layer_t^s$ and $1 \leq k \leq n$. The set of all transformations for a source metamodel s and a target metamodel t is called $Transformation_t^s$.

We naturally extend the notion of union in definition 2 to models (definition 5), match-apply models (definition 6) and transformation rules (definition 7).

3.3 Transformation collapse definitions

Let us now define some useful functions for the construction of a transformation's state space. The Graph Node Collapse function allows merging two nodes of a graph having the same type. This function is subsequently used by the Graph Collapse Function that recursively builds a set of all the possible collapsed graphs from a graph.

Definition 10. *Graph Node Collapse*
Let $\langle V, E, \tau \rangle \in TG$ be a typed graph. A graph node collapse is a function $\chi : TG \rightarrow \mathcal{P}(TG)$ such that:

$$\begin{aligned}\chi_{\langle V,E,\tau\rangle} = \big\{ \langle V \backslash \{y\}, E', \tau \backslash (y, \tau(y)) \rangle \mid \\ x, y \in V \ \wedge \ \tau(x) = \tau(y) \wedge \\ E' = \{(x, z) \mid (y, z) \in E\} \cup \\ \{(z, x) \mid (z, y) \in E\} \cup \\ \{(w, z) \mid (w, z) \in E \wedge w \neq y \wedge z \neq y\} \big\}\end{aligned}$$

This definition is naturally extended to transformations TR_t^s by limiting the two elements x and y that are collapsed to be either members of the $Match$ pattern of the transformation or elements that are connected by a backward link*.*

Definition 11. *Graph Collapse Function*
Let $g \in TG$ be a typed graph. The graph collapse function $collapse : TG \rightarrow \mathcal{P}(TG)$ is recursively defined as:

$$collapse(g) = \begin{cases} \{g\} & if\ \chi_g = \emptyset \\ \chi_g \ \cup \ \{g\} \ \cup \ \bigcup_{g' \in \chi_g} collapse(g') & if\ \chi_g \neq \emptyset \end{cases}$$

This definition is also naturally extended to transformation rules TR_t^s.

Proposition 1. *Finiteness of the result of the graph collapse function*
Let $\langle V, E, \tau \rangle \in TG$ be a typed graph. The collapsed graph set $collapse(\langle V, E, \tau \rangle)$ is a finite set of graphs, each graph in that set having a finite set of nodes.

3.4 State space

In order to define the state space for a transformation let us start by defining the possible combinations of transformations within a layer. More than that, we also define a label for each of those combinations of transformation which is used as label for the transitions in the transformation state space we build. These labels hold the identifiers of the transformations leading to a state and will be subsequently used to build counterexamples for properties that are unsatisfiable.

Definition 12. *Layer combinations*

Let $l \in Layer^s_t$ be a layer. The set of layer combinations CL_l is obtained as follows:

$$CL_l = \bigcup_{tc \in \mathcal{P}(l)} (tc, \bigsqcup_{t \in tc} t)$$

Definition 13. *Transformation state space*

Let $tr = [l_1 :: \ldots :: l_n] \in Transformation^s_t$ be a transformation. The transformation state space $SP_{tr} \subseteq TR^s_t \times (\mathcal{P}(TR^s_t) \times \mathbb{N}) \times TR^s_t$ is the least set that satisfies the following rules:

$$\frac{(tc, ut) \in CL_{l1}, tr = [l_1 :: R] \in Transformation^s_t, st \in collapse(ut)}{\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{tc_1} st \in SP_{tr}}$$

$$\frac{\begin{array}{c} tr = [H :: l_k :: l_{k+1} :: R] \in Transformation^s_t, st \xrightarrow{tc_k} st' \in SP_{tr} \\ tc \in \mathcal{P}(l_k), (tc', ut) \in CL_{l_{k+1}}, st'' \in collapse(st' \sqcup ut) \mid st' \end{array}}{st' \xrightarrow{tc'_{k+1}} st'' \in SP_{tr}}$$

Notice that H and R are lists. We also define SP^*_{tr} as the transitive closure of SP_{tr}. The $\mid : \mathcal{P}(TR^s_t) \times TR^s_t \rightarrow \mathcal{P}(TR^s_t)$ operator enforces that the backward links existing in the second parameter transformation also exist in the transformations of the first parameter.

We now build the state space for a transformation by gathering all the combinations of transformations for each layer, the result of collapsing them, and building the state space as shown in Fig. 5. Notice in particular that the second inference rule in definition 13 merges the states from a previous layer k and from the current layer $k + 1$. Notice also that all transitions in the transition state space are labeled with the transformations tc_k from the previous k layer that caused it.

Proposition 2. *Finiteness of the transformation state space*

Let $[l_1 \ldots l_n] \in Transformation^s_t$ be a transformation. The transformation state space $SP_{[l_1 \ldots l_n]}$ is finite.

The result in proposition 2 is crucial since by definition model checking can only be performed on finite state spaces.

3.5 Property semantics

Let us now proceed to formally define the semantics of our properties in the state space generated by the rules of definition 13. As we have stated in section 2, a property can be *satisfiable*, *unsatisfiable* or *non provable*. We start with the definition of a state in a state space (formally defined as a transformation) being model of a property. As a reminder, each state of the state space is a symbolic representation of a set of models given as input to the transformation being validated and their corresponding transformations. In fact, a state holds a set of patterns that should be instantiated in the input model — the *match* part of the state — as well as in the output model — the *apply* part of the state. By validating a property at the level of the symbolic states, we validate it for the whole set of input and output models of a given transformation.

Definition 14. *Model of a Property*
A transformation rule $\langle V_r, E_r, \tau_r, Match_r, Apply_r, Il_r\rangle = T \in TR^s_t$ *is a model of a property* $\langle V_p, E_p, \tau_p, Match_p, Apply_p, Il_p\rangle = P \in Property^s_t$*, written* $T \models^s P$ *if:*

1. $\langle V_p, E_p \setminus Il_p, \tau_p\rangle$ *is a typed subgraph of* $\langle V_r, E_r, \tau_r\rangle$
2. *if* $v_p \to v'_p \in Il_p$ *then there exists* $v_r \to v'_r \in E^*_r$ *where* $\tau(v_p) = \tau(v_r)$, $\tau(v'_p) = \tau(v'_r)$ *and* E^*_r *is obtained by the transitive closure of* E_r*.*

Definition 15. *Satisfiable Property*
Let $tr = [l_1 :: \ldots :: l_n] \in Transformation^s_t$ *be a transformation.* tr satisfies *property* $P \in Property^s_t$*, written* $tr \models P$*, where:*

$$tr \models P \Leftrightarrow \forall s_0 \xrightarrow{lb_0} \ldots \xrightarrow{lb_n} s_n \in SP^*_{tr} \,.\, (\exists i \,.\, s_i \models^s match(P)) \Rightarrow (\exists j \geq i \,.\, s_j \models^s P)$$

$$\text{where } s_0 = \langle\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset\rangle \text{ and } 0 \leq i \leq j \leq n.$$

Informally, for all paths belonging to tr's state space, if the property's *match* pattern is found in a given state, then a subsequent state in that path is model of the property. Note that the projection function *match* returns the match pattern of a property.

Definition 16. *Unsatisfiable Property*
Let $tr = [l_1 :: \ldots :: l_n] \in Transformation^s_t$ *be a transformation.* $tr \in TR$ *does* not satisfy *property* $P \in Property^s_t$*, written* $tr \not\models P$*, where:*

$$tr \not\models P \Leftrightarrow \exists s_0 \xrightarrow{lb_0} \ldots \xrightarrow{lb_n} s_n \in SP^*_{tr} \,.\, (\exists i \,.\, s_i \models^s match(P)) \Rightarrow (\nexists j \geq i \,.\, s_j \models^s P)$$

$$\textit{where } s_0 = \langle\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset\rangle \textit{ and } 0 \leq i \leq j \leq n.$$

The sequence $lb_0, \ldots, lb_n$ *is called a* counterexample *for property* P *in transformation* tr*.*

Informally, there exists a path belonging to tr's state space where the property's *match* pattern is found in a given state, but no subsequent state in that path is model of the property.

Definition 17. *Non Provable Property*

Let $tr = [l_1 :: \ldots :: l_n] \in Transformation_t^s$ *be a transformation. A property* $P \in Property_t^s$ *is* not provable *for* tr*, written* $tr \not\Vdash P$*, where:*

$$tr \not\Vdash P \Leftrightarrow \forall s_0 \xrightarrow{lb_0} \ldots \xrightarrow{lb_n} s_n \in SP_{tr}^* \,.\, (\nexists i \,.\, s_i) \vDash^s match(P)$$

$$where\ s_0 = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle\ and\ 0 \leq i \leq n.$$

Again informally, the *match* pattern can never be found in any state of the state space of tr.

4 Experimentation and Results

Using our implementation — downloadable at [7] — in SWI-Prolog, we have generated a state space for the the presented police station transformation, resulting in a state space with an order of magnitude of 10^4 states. Our implementation reflects the formalization in section 3. The transformation description in DSLTrans is represented as a set of facts in a entity/relationship schema, and the generated state space is represented as a list of transition predicates *t(LayerId,SquareGraphComb, Label, SquareGraphComb')*. The layer identifier *LayerId* precisely identifies the depth position of each of the transition's states, in the overall state space. Each state *SquareGraphComb* and *SquareGraphComb'* is represented as predicate *graph(match(Match),apply(Apply), blinks(BLinks))*, where *Match*, *Apply* and *BLinks* are lists of entities and relations which were merged and combined from the given transformation description.

5 Conclusions and Future Work

In this paper we have presented a model checker for model transformations expressed in the DSLTrans language. The transformations in DSLTrans are by construction confluent and terminating [3]. We have added to the language the possibility to establish syntactic structural correspondences between patterns in the source language and patterns in the target language of the transformation. This correspondence, which we call properties, is checked in a finite state space which is generated by all the possible combinations of applications of the rules specified in the transformation. Once one such property is validated for the transformation at the meta level, we can certify that it holds for all input instances of that transformation. As future work, we will perform experiments on larger transformation and address spatial and time complexities in our state space generation algorithm. Given that, on average, many states for a given state space share the same structure, we are considering using BDD-like structures [4] to compact space and accelerate state space calculation and property proof. Another possibility is to use available model checkers as interpreters for our algorithm. In this fashion we could benefit from already studied state space explosion control mechanisms. Finally, the study presented in this paper needs to be extended to structures with more semantic content than the one that can be represented by plain typed graphs. With this work we have made significant progress in understanding the fundamental issues in building a model checker for model transformations. However, a more detailed understanding and formalization of the semantics of metamodels, models and properties is needed in order to build proofs at the level of abstraction a transformation engineer would require.

References

1. Kyriakos Anastasakis, Behzad Bordbar, and Jochen Küster. Analysis of model transformations via alloy. In B. Baudry, A. Faivre, S. Ghosh, and A. Pretschner, editors, *Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2007), Nashville, TN (USA)*, pages 47–56, Berlin/Heidelberg, October 2007. Springer.
2. Mark Asztalos, Laszlo Lengyel, and Tihamer Levendovszky. Towards automated, formal verification of model transformations. In *ICST 2010: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 15–24. IEEE Computer Society, 2010.
3. Bruno Barroca, Levi Lucio, Vasco Amaral, Roberto Felix, and Vasco Sousa. A visual language for model transformations. Technical report, UNL-DI-2-2010, University Nova de Lisboa, Portugal, 2010. `http://solar.di.fct.unl.pt/twiki/pub/BATICCCS/ModelTransformationPapers/vltechrep.pdf`.
4. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
5. Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel matching for automatic model transformation generation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Volter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2008.
6. SOLAR Group. Detailed proofs for the paper: "a technique for automatic validation of model transformations". `http://solar.di.fct.unl.pt/twiki/pub/BATICCCS/ModelTransformationPapers/detailed_proofs.pdf`.
7. SOLAR Group. Transformation model checker. `http://solar.di.fct.unl.pt/twiki/pub/BATICCCS/ReleaseFiles/transmc.zip`.
8. SOLAR Group. Transforming uml to java using dsltrans. `http://solar.di.fct.unl.pt/twiki/pub/BATICCCS/ModelTransformationPapers/UML2Java.zip`.
9. Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.
10. Anantha Narayanan and Gabor Karsai. Verifying model transformations by structural correspondence. *ECEASST*, 10, 2008.
11. Object Management Group. Query/view/specification, December 2005. `http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf`.