

# Symbolic Verification of Translation Model Transformations

Levi Lúcio<sup>†</sup>, Bentley James Oakes<sup>†</sup>, and Hans Vangheluwe<sup>‡†</sup>

<sup>†</sup>School of Computer Science, McGill University, Canada

<sup>‡</sup>University of Antwerp, Belgium

{levi, hv}@cs.mcgill.ca, bentley.oakes@mail.mcgill.ca

**Abstract.** As model transformations are a required part of model-driven development, it is crucial to provide techniques that address their formal verification. One approach that proven very successful in program verification is *symbolic execution*. The symbolic abstraction in these techniques allows formal properties to be exhaustively proved for all executions of a given program. In our approach we apply the same abstraction principle to verify model transformations. Our algorithm builds a finite set of path conditions which represents all concrete transformation executions through a formal abstraction relation. We are then able to prove properties over all transformation executions in a model-independent way. This is done by examining if any created path condition violates a given property, which will produce a counterexample if the property does not hold for the transformation. Results are presented here for our implementation which suggests that our approach is feasible and can scale to real-world transformations.

**Keywords:** Model Transformations, Symbolic Verification, Translation

## 1 INTRODUCTION

Model transformations were described as the *heart and soul* of model driven software development (MDD) by Sendall and Kozaczynski in 2003 [1]. Due to their practicality and appropriate level of abstraction, model transformations are the current technique for performing computations on models. In their well-known 2006 paper ‘A Taxonomy of Model Transformations’, the authors Mens, Czarnecki and Van Gorp call for the development of verification, validation and testing techniques for model transformations [2]. 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 Amrani, Lúcio *et al.* in [3].

In this paper, we present our work on verification of properties on model transformations. Specifically, we discuss concrete algorithms that can prove whether properties will hold or do not hold on all executions of a transformation written in the DSLTrans transformation language. Properties are proved through a process that symbolically constructs a set of path conditions, where each path condition represents an infinite number of concrete transformation executions through an *abstraction relation*.

In our previous work [4], this property-proving algorithm was presented as a proof-of-concept. In the present work, we significantly expand that proof-of-concept by clarifying and offering discussions on validity and completeness for the presented algorithms. We also provide an implementation that we believe will scale to industrial applications, as validated by an automotive case study described in [5].

Our specific contributions include:

- An algorithm for constructing all path conditions for a given DSLTrans transformation;
- An algorithm that proves transformation properties over these path conditions using an abstraction relation;
- Validity and completeness proofs of the path condition construction and property proving algorithms;
- A discussion of performance and scalability results for our implementation.

Our approach is feasible due to the use of the transformation language DSLTrans, formally introduced in [6]. Briefly, 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 [7–9] that DSLTrans is sufficiently expressive to tackle typical translation problems. This sacrifice of Turing-completeness allows us to construct a provably-finite set of path conditions, as described in [4]. Our approach considers a core subset of the DSLTrans language that does not include negative conditions in rules or attribute manipulation.

Informed by the structure of DSLTrans transformations, our approach defines an algorithm for the creation of path conditions. Each path condition that is created represents a set of concrete transformation executions through an *abstraction relation* that we formally define. Once the set of all path conditions has been created for the transformation, we can then prove structural *model syntax relations* [3] using this relation. Such properties are essentially pre-condition/post-condition axioms involving statements about whether certain elements of the input model have been transformed into elements of the output model. Several authors have explored this kind of transformation properties [10–13]. The properties examined can be proven to hold for all

executions of a given model transformation, no matter the input model. Therefore, our proof technique is *transformation dependent* and *input independent* as classified by [3].

Our methods differ from previous work in the transformation verification field in that we require no intermediate representation for a specific proving framework, as in for example [14–16], but instead work on DSLTrans transformations themselves. Along with DSLTrans rules, all of the constructs involved in our algorithms are graph-based. This intuitive representation allows for relatively simple steps in our property proving algorithm, as the metamodels, models, and properties involved are all constructed in similar graphical representation. This also provided us with performance gains over our previous implementation, due to the use of efficient graph-matching libraries to manipulate the graph structures required.

A large difficulty in any exhaustive proof technique is the tendency for the state space to explode, even when abstractions are performed to render the search space finite. A later section of this work will discuss optimisation opportunities and performance results obtained from our implementation. The scalability of our approach will also be analysed in order to infer the algorithm’s potential applicability to real-world problems. A real-world industrial case study will also be briefly presented.

This paper is organised as follows: formal background for this work is presented in Section 3, while Section 2 briefly introduces the DSLTrans model transformation language and its formal semantics. The algorithms to build the complete set of path conditions for the transformation will be discussed in Section 4. In Section 6, the process for proving properties is presented, along with an introduction of our property language. Formal proofs for the validity and completeness of our work will be presented in both Section 4 and Section 6

In Section 7, we introduce our implementation with sample scalability results; Section 9 presents the related work; and finally in Section 10 we conclude with remarks and future work.

## 2 The DSLTrans Transformation Language

In this section we will introduce the DSLTrans transformation language and its constructs from [6]. A formal treatment of the syntax and semantics of DSLTrans is found in Section 3.1.

A DSLTrans transformation has a source and a target metamodel, which are seen in Figure 1<sup>1</sup>. This *Police Station* transformation will be presented throughout the rest

---

<sup>1</sup> The models presented have been created in the DSLTrans Eclipse plug-in [8].

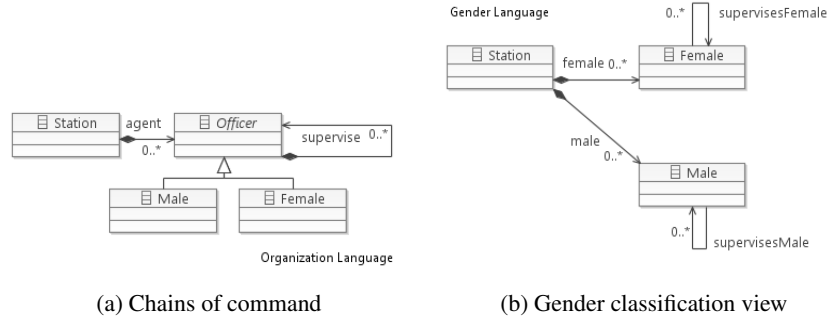
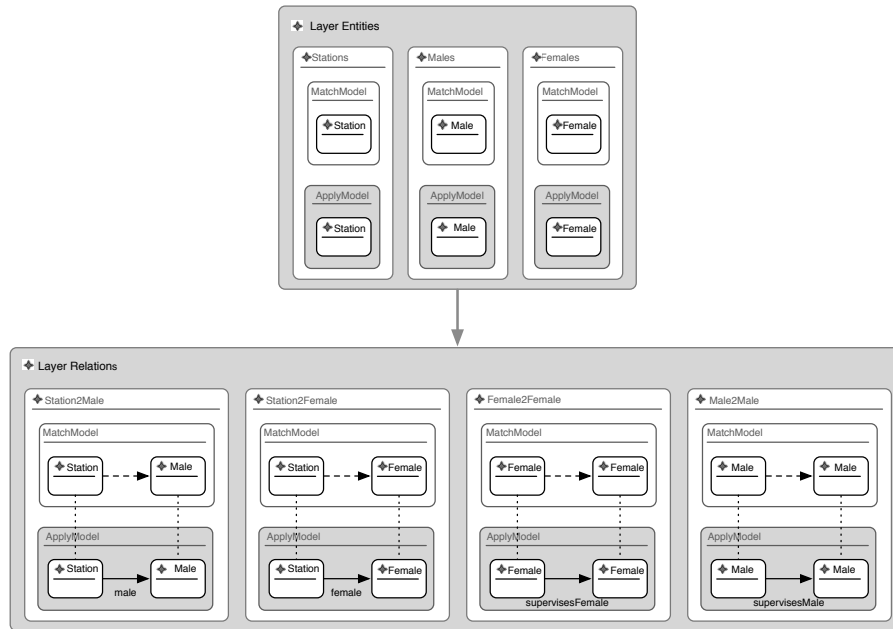


Fig. 1: Metamodels for the police station

of this paper as an example transformation. The metamodel in Figure 1a represents a language for describing the chain of command in a police station, which includes the male (*Male* class) and female officers (*Female* class). The metamodel in Figure 1b 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.

Fig. 2: The *Police Station* model transformation expressed in DSLTrans



In Figure 2 we present a transformation written in DSLTrans between models of both languages. A description of relevant constructs as well as visual notation remarks

are found in Section 2.2. Note that the transformation is formed from layers where each layer is a set of transformation rules. The transformation will execute layer-by-layer, where transformation rules in a layer will execute in a non-deterministic order but must produce a deterministic result, due to the fact that DSLTrans is confluent by construction [6].

Another important characteristic of DSLTrans transformations is that they are not Turing-complete. As discussed in [6], non-completeness is required to make the rule sequences finite, but yet still allows for appropriate expressiveness.

Besides the fact that DSLTrans' transformations are free of constructs that imply unbounded recursion or non-determinism, DSLTrans' transformations are strictly out-place, meaning no changes are allowed to the input model. However, the output metamodel for a DSLTrans transformation can 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 [17], as no deletion of output elements is strictly required.

Each individual transformation rule is composed of two graphs. The first graph is denoted as the *match graph*, and is a pattern holding elements from the source metamodel. Likewise, the *apply graph* is a pattern containing elements from the target metamodel. These patterns are formally defined in ?? below.

Note that in our approach, we require that the match graph of a rule is not a subset of the match graph of any other rule. This must also take into account transitive links found in the rules. This requirement is to prevent ordering of rules, where a rule cannot execute independently of other rules. This is undesirable for the algorithm as presented here. However, as seen in [Bentley](#)  [Cite ICGT paper](#) , the expressiveness of transformations is not restricted. In that work, we detail a rule processing step to handle overlapping rules.

For example, consider the transformation rule marked *Stations* in the first rule layer in Figure 2. The match graph holds one *Station* element from the source metamodel, while the apply graph holds one *Station* element from the target metamodel. This means that for all elements in the input model which are of type *Station* in the *Organization Language*, a element of type *Station* in the *Gender Language* will be created in the output model.

The purpose of this *Police Station* transformation is to flatten a chain of command given in the *Organization language* into two independent sets of male and female officers represented in the *Gender language*. The command relations will be kept during this transformation, i.e. a female officer will have a direct association to all her female

subordinates and likewise for male officers. Note that differences in the gender classification metamodel means some relations present in the input model will not be retained in the output model.

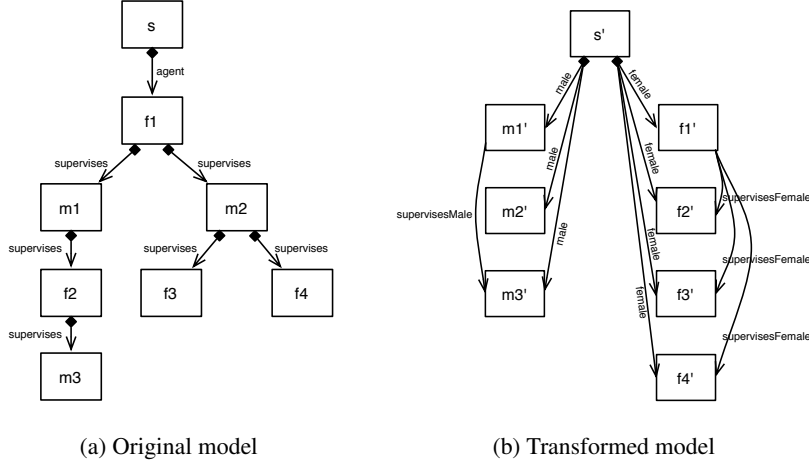


Fig. 3: Model before and after transformation

An example of this transformation's execution can be observed in Figure 3, where the input model is on the left and the output model is on the right. Notice that the elements  $s$ ,  $m_k$  and  $f_k$  in Figure 1a are instances of the source *Organization* metamodel elements *Station*, *Male* and *Female* respectively. The primed elements in Figure 1b are their counterpart instances in the target *Gender* metamodel.

## 2.1 Properties to Prove

The properties we aim to prove on the Police Station transformation are structural properties. These properties are composed of a pre-condition and a post-condition component, as seen in Figure 4.

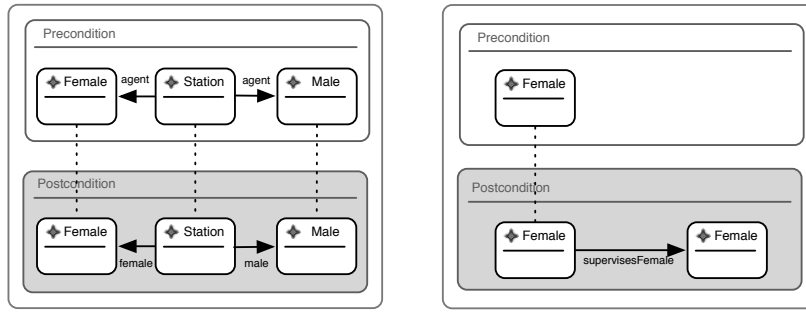
Informally, the property can be read as 'if the pre-condition graph matches on the input model to the transformation, then the post-condition graph will match any output model produced. Further details as well as formal validity and completeness are discussed in Section 6.

As a brief example, consider the property in Figure 4a. The pre-condition graph is composed of a Station element connected to a Female element and a Male element, where all elements are from the *Organization language* metamodel. This structure is repeated in the post-condition graph, with the difference that the metamodel for these

elements is the *Gender language*. Thus, this property represents the statement “*a model which includes a police station that has both male and female officers will be transformed into a model where the male officer will exist in the male set and the female officer will exist in the female set*”. We expect this property to always hold in our transformation.

In contrast, we do not expect the property in Figure 4b to always hold. This property represents the statement “*any model which includes a female officer will be transformed into a model where that female officer will always supervise another female officer*”. It is not difficult to construct an input model where the pre-condition holds, but the post-condition does not – an input model with one female officer.

These properties will be presented again in Section 6 to discuss our property proving technique. Following this, experiments in Section 8 will present timing results from proving these properties on the Police Station transformation.



(a) Property 1 – Expected to hold

(b) Property 2 – Not expected to hold

Fig. 4: Properties to be proved on the Police Station transformation

## 2.2 DSLTrans Constructs

This section will describe all of the DSLTrans constructs involved in our property-proving algorithm. We will illustrate the constructs by referring to the transformation in Figure 2. Formal details can be found in Section 3.1 as well as [6].

- **Match Elements:** Match elements are variables typed by elements of the source metamodel which will match over elements of that type (or subtype) in the input model when the transformation is executed. Note that match elements in a rule are searched for injectively in a model. This means that, for example, if a match graph

includes two elements of type *Station*, then the rule will only match over models that include at least two instances of type *Station*.

In the DSLTrans notation as seen throughout this paper, the match elements will be in a white box in the top half of a rule.

- **Direct Match Links:** Direct match links are variables typed by labelled associations of the source metamodel, which will match over associations of the same type 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. Our notion of indirect links captures only acyclic EMF containment associations. As such, it avoids cycles and infinite amounts of matches over the transitive closure of associations in the input models.  
In Figure 2, indirect match links are represented in all the transformation rules in the last layer as dashed arrows between elements in the match graph.
- **Backward Links:** Backward links connect elements of the match and the apply patterns of a DSLTrans rule in order to represent dependencies on element creation by previous layers of the transformation. Note that the only possibility to reuse elements created from a previous layer is to refer to them using backward links. Backward links are found in Figure 2 in all transformation rules on the last layer and are depicted as dashed lines.
- **Apply Elements and Apply Links:** Apply elements and apply links are similar to match elements and match links, but are instead typed by elements of the target 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 graph of the rule is isomorphically found in the input model.

Consider the transformation rule denoted *Station2Male* in the last rule layer of Figure 2. This rule takes *Station* and *Male* elements of the *Gender Language* metamodel, where these elements were created in a previous layer from *Station* and



Male elements of the *Organization Language* metamodel, and connects them using a *male* association.

### 3 Formal Background

In this section we will introduce a few graph concepts that will be used throughout all this paper. The concepts are introduced in an abstract manner as they are well known from graph theory and only slightly customized for our purposes.

We will start by introducing the notion of typed graph. A typed graph is the essential object we will use throughout our mathematical development. Typed graphs will be used to formalise all the important graph-like structures we will present in this paper. A typed graph is a directed multigraph (a graph allowing multiple edges between two vertices) where vertices and edges are typed.

**Definition 1.** *Typed Graph*

A typed graph is a 4-tuple  $\langle V, E, (s, t), \tau \rangle$  where:  $V$  is a finite set of vertices;  $E$  is a finite set of edges connecting the vertices  $V$ ;  $(s, t)$  is a pair of functions  $s : V \rightarrow E$  and  $t : V \rightarrow E$  that respectively provide the source and target vertices for each edge in the graph; function  $\tau : V \cup E \rightarrow VT \cup ET$  is a typing function for the elements of  $V$  and  $E$ , where  $VT$  and  $ET$  are disjoint finite sets of vertex and edge type identifiers and  $\tau(v) \in VT$  if  $v \in V$  and  $\tau(e) \in ET$  if  $e \in E$ . Edges  $e \in E$  are noted  $v \xrightarrow{e} v'$  if  $s(e) = v$  and  $t(e) = v'$ , or simply  $e$  if the context is unambiguous. The set of all typed graphs is called TG.

We now define how two typed graphs are united. A union of two typed graphs is trivially the set union of all the components of those two typed graphs. Note that we do not require the components of the two graphs to be disjoint, as in the following joint unions will be used to merge typed graphs.

**Definition 2.** *Typed Graph Union*

Let  $\langle V, E, (s, t), \tau \rangle, \langle V', E', (s', t'), \tau' \rangle \in \text{TG}$  be typed graphs. The typed graph union is the function  $\sqcup : \text{TG} \times \text{TG} \rightarrow \text{TG}$  defined as:

$$\langle V, E, (s, t), \tau \rangle \sqcup \langle V', E', (s', t'), \tau' \rangle = \langle V \cup V', E \cup E', (s \cup s', t \cup t'), \tau \cup \tau' \rangle$$

We are interested in relations between typed graphs that are structure-preserving, i.e. homomorphisms. Homomorphisms between typed graphs preserve not only structure, but also the types of vertices and edges that are mapped.

**Definition 3. Typed Graph Homomorphism**

Let  $\langle V, E, st, \tau \rangle = g$  and  $\langle V', E', st', \tau' \rangle = g' \in \text{TG}$  be typed graphs. A typed graph homomorphism between  $g$  and  $g'$  is a function  $f : V \rightarrow V'$  such that for all  $v_1 \xrightarrow{e} v_2 \in E$  we have that  $f(v_1) \xrightarrow{e'} f(v_2) \in E'$  where  $\tau(v_1) = \tau'(f(v_1))$ ,  $\tau(v_2) = \tau'(f(v_2))$  and also  $\tau(e) = \tau'(e')$ . When an injective typed graph homomorphism exists between  $g$  and  $g'$  we write  $g \triangleleft g'$ . When a surjective typed graph homomorphism exists between typed graphs  $g$  and  $g'$  we write  $g \blacktriangleleft g'$ .

Note that, trivially, a typed graph homomorphism is a graph homomorphism.

We now define the useful notion of typed subgraph. As expected, a typed subgraph is simply a restriction of a typed graph to some of its vertices and edges.

**Definition 4. Typed Subgraph**

Let  $\langle V, E, st, \tau \rangle = g$ ,  $\langle V', E', st', \tau' \rangle = g' \in \text{TG}$  be typed graphs.  $g'$  is a typed subgraph of  $g$ , written  $g' \sqsubseteq g$ , iff  $V' \subseteq V$ ,  $E' \subseteq E$  and  $\tau' = \tau|_{V' \cup E'}$ .

Two typed graphs are said to be isomorphic if they have exactly the same shape and related vertices and edges have the same type.

**Definition 5. Typed Graph Isomorphism**

Let  $\langle V, E, st, \tau \rangle = g$ ,  $\langle V', E', st', \tau' \rangle = g' \in \text{TG}$  be typed graphs.  $g$  and  $g'$  are isomorphic, written  $g \cong g'$ , if and only if there exists a bijective typed graph homomorphism  $f : V \rightarrow V'$  such that  $f^{-1} : V' \rightarrow V$  is a typed graph homomorphism.

**Levi** ► **Make sure that all graphs that need to be disjoint are disjoint during the mathematical development** ◀

**Notation** In the mathematical development that follows we will often use a ‘dot’ notation to represent that we do not care about the value of a particular variable in a given context. For example, assuming  $p$  is a predicate of arity 2, we will write  $p(\cdot, b)$  to signify that while we capture values of interest in variable  $b$ ,  $p$ ’s first argument is disregarded.

**3.1 Formalization of Required Constructs**

This section will formally describe the DSLTrans constructs that are required in our property-proving algorithm.

The theory is based on the notion of typed graphs as described in Section 3.

We will start by introducing the notion of *metamodel*, which in DSLTrans is used to type the input and output models of a DSLTrans transformation.

**Definition 6. Metamodel**

A metamodel is a 5-tuple  $\langle V, E, st, \tau, \leq \rangle$  where  $\langle V, E, st, \tau \rangle \in \text{TG}$  is a typed graph,  $(V, \leq)$  is a partial order and  $\tau$  is a bijective typing function. Additionally we also have that: if  $v \in V$  then  $\tau(v) \in VTypes \times \{abstract, concrete\}$  where  $VTypes$  is the set of vertex type names; if  $e \in E$  then  $\tau(e) \in ETypes \times \{containment, reference\}$ , where  $ETypes$  is a set of edge type names. The set of all metamodels is called **META**.

A formal metamodel is particular kind of typed graph where vertices represent classes and edges relations between those classes. A typed graph representing a metamodel has two special characteristics: on the one hand, the typing function for vertices and edges is bijective. This means that each type occurs only once in the metamodel, as is to be expected. On the other hand a metamodel is equipped with a partial order between vertices. This partial order is used to model specialization at the level of the metamodel's classes. Note that here we have overridden the co-domain of the typing function in the original typed graph presented in Definition 1 in order to allow distinguishing between *abstract* and *concrete* classes, as well as between *containment* and *reference* edges in our metamodels.

**Levi** ► **Cardinalities are not treated, meaning all relations can be n-to-m. It may be necessary to treat cardinalities.** ◀

**Definition 7. Metamodel Instance**

An instance of a metamodel  $mm = \langle V', E', st', \tau', \leq \rangle \in \text{META}$  is a typed graph  $\langle V, E, st, \tau \rangle \in \text{TG}$ , where the codomain of  $\tau$  equals the codomain of  $\tau'$ . Also, there is a typed graph homomorphism  $f : V \rightarrow V'$  from  $\langle V, E, st, \tau \rangle$  to metamodel  $mm^*$  such that the graph  $\langle V, \{v \rightarrow v' \in E \mid \tau(v \rightarrow v') = (\cdot, containment)\}, st \rangle$  is acyclic. The set of all instances for a metamodel  $mm$  is called  $\text{INSTANCE}^{mm}$ .

A metamodel instance is a useful intermediate formal notion that lies between metamodel and model. The injective typed graph homomorphism between a metamodel instance and metamodel models multiple “instances” of objects and relations being typed by one single class or relation of the metamodel. Metamodel instances do not allow the transitive closure of containment relations, as enforced by EMF.

**Definition 8. Containment Transitive Closure**

The containment closure of a metamodel instance  $\langle V', E', st', \tau' \rangle \in \text{INSTANCE}^{mm}$  is a typed graph  $\langle V, E, st, \tau \rangle$  where we have that  $V = V'$ ,  $\tau \subseteq \tau'$  and  $\tau$ 's codomain is the union of the codomain of  $\tau'$  and the set  $\{indirect\}$ . We also have that  $E' = E \cup E_c^*$  where  $E_c^*$  is the transitive closure of the set  $\{v \xrightarrow{e} v' \mid \tau(v \xrightarrow{e} v') = (\cdot, containment)\}$  and

if  $e \in E \setminus E'$  then  $\tau(e) = \text{indirect}$ . We note  $mi^*$  the containment closure of a metamodel instance  $mi \in \text{INSTANCE}^{mm}$ .

Given a metamodel instance, its containment transitive closure includes, besides the original graph, all the edges belonging to the transitive closure of containment links in that metamodel instance. The transitive edges are typed as *indirect*.

In definitions that follow we will use the  $*$  notation, as in Definition 31, to denote the containment transitive closure of structures that directly or indirectly include metamodel instances. For example,  $tg^*$  would represent the containment transitive closure of typed graph  $tg$  wherever containment edges are found in the graph.

**Definition 9. Model**

A model of a metamodel  $mm = \langle V', E', st', \tau', \leq \rangle \in \text{META}$  is a metamodel instance  $\langle V, E, st, \tau \rangle \in \text{INSTANCE}^{mm}$ , such that: there exists an injective typed graph homomorphism  $f : V \rightarrow V'$  from  $\langle V, E, st, \tau \rangle$  to metamodel  $mm^*$  where, if there exists an edge  $f(a) \xrightarrow{e'} b \in E'$  where  $\tau(e') = (\cdot, \text{containment})$ , then we also have that  $f(b) \xrightarrow{e} c \in E$  and that  $f(c) = b$ . The set of all models for a metamodel  $mm$  is called  $\text{MODEL}^{mm}$ .

A model, as per Definition 32, is a metamodel instance where all the containment relations are respected. This means that if an object having a containment relation exists in the model, then the model will also contain an instance of that containment relation together with a contained object.

**Levi** ► **this only works if 0 multiplicities are allowed for the metamodel example in the paper.** ◀

**Definition 10. Metamodel Pattern and Indirect Metamodel Pattern**

A pattern of a metamodel  $mm \in \text{META}$  is an instance of  $mm$ . Given a metamodel pattern  $\langle V', E', st', \tau' \rangle \in \text{INSTANCE}^{mm}$  we have that  $\langle V, E, st, \tau \rangle$  is an indirect pattern if  $V = V'$ ,  $E \subseteq E'$ , the codomain of  $\tau$  is the union of the codomains of  $\tau'$  and the set  $\{\text{indirect}\}$ . Also, if  $v_1 \xrightarrow{e} v_2 \in E \setminus E'$ , then we have that  $\tau(e) = \text{indirect}$ . Given a metamodel  $mm$ , the set of all metamodel patterns for  $mm$  is called  $\text{PATTERN}^{mm}$ . The set of all indirect metamodel patterns for  $mm$  is called  $\text{IPATTERN}^{mm}$ .

Metamodel patterns are introduced in Definition 34 as an auxiliary notion, distinct from the notion of metamodel instance (although formally they are the same), with the goal of introducing new structures that are used to represent rules (and later on path conditions and properties). An *indirect* metamodel pattern is a metamodel pattern that includes edges typed as *indirect*, used in the coming mathematical development to model DSLTrans rules.

**Definition 11. Transformation Rule**

A transformation rule is a 6-tuple  $\langle V, E, (s, t), \tau, Match, Apply \rangle$ , where:  $Match = \langle V', E', st', \tau' \rangle \in \text{IPATTERN}^{sr}$  such that  $Match \neq \varepsilon^2$  is a non-empty indirect metamodel pattern;  $Apply = \langle V'', E'', st'', \tau'' \rangle \in \text{PATTERN}^{tg}$  such that  $Apply \neq \varepsilon$  is a metamodel pattern. We also have that  $V = V' \cup V''$ ,  $E \subseteq E' \cup E''$  and  $\tau \subseteq \tau' \cup \tau''$ , where the codomain of  $\tau$  is the union of the codomains of  $\tau'$  and  $\tau''$  and the set  $\{backward\}$ . An edge  $e \in E \setminus E' \cup E''$  is called a backward link and is such that  $s(e) \in V''$ ,  $t(e) \in V'$  and  $\tau(e) = backward$ . We additionally impose that there always exists a  $v_1 \in V''$  in the Apply part of the rule such that  $\nexists e : v_1 \xrightarrow{e} v_2$  and  $\tau(e) = backward$ . The set of all transformation rules for a source metamodel  $sr$  and a target metamodel  $tg$  is called  $\text{RULE}_{tg}^{sr}$ .

A transformation rule includes a non-empty match pattern and a non-empty apply pattern (also known in the model transformation literature as a rule's *left hand side* and *right hand side*). The apply pattern of a rule always contains at least one apply element that is not connected to a backward link, meaning in practice that a rule will always produce something and not only match. A match pattern can include indirect links that are used to transitively match a model. An apply pattern does not include indirect links as it is used only for the construction of parts of instances of a metamodel. A transformation rule includes backward links, as informally introduced in Section 2.2. Backward links are formally typed as *backward*.

**Definition 12. Backward Matcher for a Transformation Rule**

Let  $rl = \langle V, E, st, \tau, Match, Apply \rangle$  be a transformation rule where  $Match = \langle V_m, E_m, st_m, \tau_m \rangle$ . We define  $rl$ 's backward matcher version, noted  $rl^{backM}$ , as the transformation rule  $\langle V', E', st', \tau', Match, Apply' \rangle \sqsubseteq rl$  where  $v_1 \xrightarrow{e} v_2 \in E'$  if and only if  $v_1, v_2 \in Match$  or  $\tau(e) = backward$ .

Definition 35 introduces the notion of backward matcher for a transformation rule which consists solely of the match pattern of a rule and its backward links, if any. The backward matcher of a rule constitutes the complete pattern that a DSLTrans rule attempts to match over an input-output model during rule execution. We will see in the next section that traceability links, generated during execution between the input and output model, are matched by transformation rules' backward links.

**Definition 13. Layer, Transformation**

A layer is a finite set of transformation rules  $l \subseteq \text{RULE}_{tg}^{sr}$ . The set of all layers for a source metamodel  $sr$  and a target metamodel  $tg$  is called  $\text{LAYER}_{tg}^{sr}$ . A transformation is

<sup>2</sup> We use the simplified  $\varepsilon$  notation to denote empty n-tuples structures.

a finite list of layers denoted  $[l_1 :: l_2 :: \dots :: l_n]$  where  $l_k \in \text{LAYER}_{ig}^{sr}$  and  $1 \leq k \leq n$ . We also impose that for any two rules  $rl_1, rl_2 \in \bigcup_{1 \leq k \leq n} l_k$  we never have that  $rl_1^{backM} \cong rl$  and  $rl \sqsubseteq rl_2^{backM}$ , or  $rl_2^{backM} \cong rl$  and  $rl \sqsubseteq rl_1^{backM}$ . The set of all transformations for a source metamodel  $s$  and a target metamodel  $t$  is called  $\text{TRANSF}_{tr}^{sr}$ .

Definition 38 formalizes the notion of a DSLTrans transformation, introduced at the beginning of this section. As expected, a formal DSLTrans transformation is composed of a sequence of layers where each layer is composed of a set of rules. The last condition of Definition 38 imposes that, for any two rules in the transformation, the backward matcher of one rule never partially or totally subsumes (or contains) the backward matcher of the other.

**Notation** In order to achieve a uniform and understandable notation throughout the text that follows we will often abbreviate input-output models (Definition 33) and transformation rules (Definition 35) to their typed graph components. We will also use a simplified notation to refer to the components of the input/output or match/apply typed graphs of those structures. For example, we will abbreviate transformation rule  $\langle V, E, st, \tau, Match, Apply \rangle \in \text{RULE}_{ig}^{sr}$  to only its typed graph  $\langle V, E, st, \tau \rangle$ . In this case we will refer to the transformation rule's vertices that belong to the match part of the graph as  $Match(V)$ , its edges as  $Match(E)$  and so on for pair  $st$  and function  $\tau$ . In a similar fashion we use the notation  $Apply(V)$ ,  $Apply(E)$ , etc. to refer to the transformation rule's components that belong to the apply part of the graph. We will also use the natural extensions of the notions of typed graph union ' $\sqcup$ ' (Definition 2), typed graph homomorphism ' $\triangleleft, \blacktriangleleft$ ' (Definition 3), typed subgraph ' $\sqsubseteq$ ' (Definition 4) and typed graph isomorphism ' $\cong$ ' (Definition 5) to input-output models and transformation rules.

**Definition 14. Model Transformation Execution**

Let  $input \in \text{MODEL}^{sr}$  and  $output \in \text{MODEL}^{tg}$  be models and  $tr \in \text{TRANSF}_{ig}^{sr}$  be a transformation. Assume we also have that:

$$\langle V, E, st, \tau, input, \epsilon \rangle, tr \xrightarrow{trstep} \langle V', E', st', \tau', input, output \rangle$$

A model transformation execution is the input-output model  $\langle V', E', st', \tau', input, output \rangle$ . The set of all model transformation executions for transformation  $tr$  is written  $\text{EXEC}(tr)$  and the empty model transformation execution is noted  $\epsilon_{ex}$ .

Finally, as stated in Definition 43, we consider a model transformation execution to be the complete input-output model resulting from executing a transformation on an

input model. Note that in this result we have the input model, the output model and the traceability links between their elements that are generated during the transformation’s execution. Although considering the complete input-output model as the transformation result is interesting for the purposes of this paper, in practice a transformation developer normally only interested in the output component of the input-output model in a transformation execution.

Note that, because in Definition 43 *output* is a formal model, we enforce that containment relations are respected in the transformation’s output.

## 4 Building Path Conditions

This section will describe how path conditions can be used to represent the executions of a model transformation, as well as the algorithm that creates the set of all possible path conditions for a given DSLTrans transformation.

### 4.1 Symbolic Execution

Our algorithm operates on the principle of symbolic execution. In order to explain the concept of symbolic execution of a transformation, let us make an analogy with program symbolic execution as introduced by King in his seminal work “*Symbolic Execution and Program Testing*” [18]. According to King, a symbolic execution of a program is a set of *constraints* on that program’s *input variables* called *path conditions*. Each *path condition* describes a traversal of the conditional branching commands of that program. A *path condition* is symbolic in the sense it *abstracts* as many concrete executions as there are instantiations of the path condition’s variables that render the path condition’s constraints true.

We can transpose this notion of symbolic execution to model transformations. The analog of an input variable in the model transformation context are *metamodel classes, relations and attributes*. As program statements impose constraints on input and output variables during symbolic execution, transformation rules impose conditions on which metamodel elements are instantiated during a concrete transformation execution, and how that instantiation happens. As well, rules in a model transformation are implicitly or explicitly scheduled. Such control and/or data dependencies must also be taken into consideration during path condition construction. Such conditions, built from transformation rules, can then be used as the analog of program *path conditions* in the context of model transformations. In fact, throughout this paper we also use the term *path condition* in the context of model transformation verification.

As in program symbolic execution, each transformation path condition *abstracts* as many concrete executions as there are input/output models that satisfy them. This is formulated as an *abstraction relation*, defined in ??.

In what follows we will examine in more detail how these symbolic execution principles can apply to the verification of model transformations.

## 4.2 Path Conditions

We shall begin illustrating the structure and purpose of path conditions by discussing rule combinations.

**Rule Combinations** A layer in a DSLTrans transformation will contain a number of rules. A set of rule combinations for a layer can then be created by taking the powerset of all rules in that layer. Each one of these rule combinations is then defined to represent all possible transformation executions where the rules in that combination would execute.

For example, in Figure 5, the rule combination marked 'AC' represents the set of transformation executions where the rules A and C would execute and no others. Another rule combination marked 'A' represents the transformation executions where only rule A would execute.

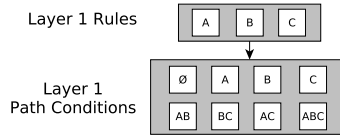


Fig. 5: Rule combinations created for a transformation layer

Note that within these rule combinations, the number of times a rule has executed is abstracted. Either a rule has executed zero times, and the rule is not represented in a rule combination, or the rule has executed some finite number of times and it is represented.

The number of rule combinations that need to be created is drastically reduced due to the semantics of DSLTrans, as described in Section 2. For instance, the fact that transformation rules in a layer will execute in a non-deterministic order but must produce a deterministic result means that rule combinations can be created instead of rule permutations. We also note that the transformation executions that these rule combinations represent are finite. This is given by the fact that DSLTrans transformations are not Turing-complete.



**Traceability and Backward Links** Before creating path conditions, the notion of rule dependency must be addressed. DSLTrans rules allow for the specification of dependencies on which elements of the output model were created from specific elements of the input model. The backward link construct in DSLTrans rules precisely defines these dependencies as further detailed in Section 2.2 and Definition 35. In order to use backward links, the traceability information of the transformation must be stored in the path conditions, mirroring the information stored during the execution of a DSLTrans model transformation [6].

In our algorithm, this traceability information is explicitly constructed to record which elements to be placed in a path condition came from the same DSLTrans rule. This information is stored in *traceability links*.

During the path condition construction process, traceability links are built for each rule as follows: for all match and apply elements of a rule, given a match element belonging to the match graph of a rule and an apply element belonging to the apply graph of the same rule, a traceability link is built between the two if the apply element is not connected to a backward link. This is intuitive: traceability links are built between a newly generated element in the output model, and the elements of the input model that originated it.

An example of the traceability link creation process is shown in Figure 6. Note that traceability links are a solid line between match and apply elements in our visual notation.

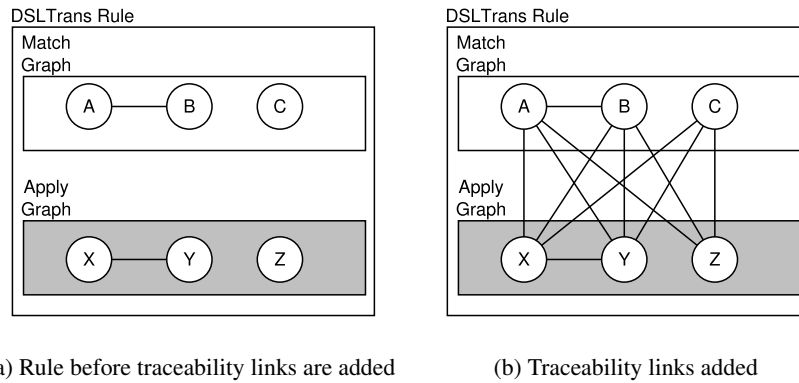


Fig. 6: Traceability links created for an abstract DSLTrans rule

Figure 7 demonstrates how this traceability information can be used for dependency specification. The rule within the figure contains a backward link, which defines the

dependency that an element of type D was created from an element of type A, and an element of type E was created from an element of type B. If this dependency is satisfied, then another element of type G should be created. This element should be associated with the E element, and have two traceability links back to the A and B elements in the match graph. Note that these backward links match over the traceability links seen between A and D, and B and E in Figure 6b.

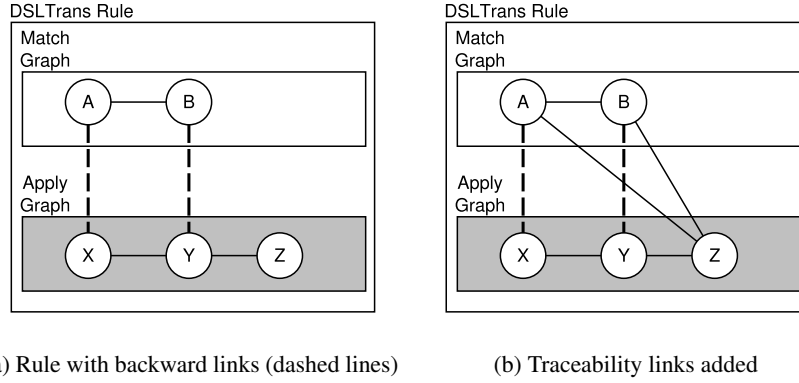


Fig. 7: Adding traceability links to an abstract DSLTrans rule with backward links

**Path Condition Creation** The structure of path conditions is similar to that of DSLTrans rules, due to their function of representing the interaction of a set of DSLTrans rules. Path conditions are composed of two graphs. The first graph represents a pattern that must be present in the input model of the transformation, while the second is a pattern which will be instantiated in the output model of the transformation, including traceability to elements matched by the rules. The formal definition of a path condition can be seen in ??.

Therefore, path condition creation from a rule combination is relatively simple. The match graph for the new path condition is created by taking a union of all the match graphs of the rules in the rule combination. Similarly, the path condition's apply graph is created by taking a union of all the apply graphs. Note that the traceability links were therefore placed to retain the information of which apply elements were generated by which match elements.

The path condition produced will then represent the same transformation executions as the rule combinations it was created from, using the abstraction relation found in ??.

**Definition 15. Path Condition**

A path condition is a 7-tuple  $\langle V, E, (s, t), \tau, Match, Apply, Rule \rangle$ , where  $Match = \langle V', E', st', \tau' \rangle \in \text{IPATTERN}^{sr}$  is an indirect pattern and  $Apply = \langle V'', E'', st'', \tau'' \rangle \in \text{PATTERN}^{tg}$  is a pattern. We also have that  $V = V' \cup V''$ ,  $E \subseteq E' \cup E''$  and  $\tau \subseteq \tau' \cup \tau''$  where the codomain of  $\tau$  is the union of the codomains of  $\tau'$  and  $\tau''$  and the set  $\{trace\}$ . An edge  $e \in E \setminus E' \cup E''$ , called a symbolic traceability link, is such that  $s(e) \in V''$  and  $t(e) \in V'$  and  $\tau(e) = trace$ . Finally, the *Rule* component in the 7-tuple is the set of rules used in the construction of the path condition, where each rule is a subgraph of  $\langle V, E, (s, t), \tau \rangle$ . The set of all path conditions for a source metamodel  $sr$  and a target metamodel  $tg$  is called  $\text{PATHCOND}_{tg}^{sr}$  and the empty path condition is noted  $\epsilon_{pc}$ .

Similarly to a transformation rule (see Definition 35), a path condition is also a typed-graph with a match and an apply part. As mentioned before, a path condition contains a combination of rules where *symbolic traceability links* represent the concrete traceability links of a transformation execution (see Definition 43). The path condition structure also contains a *Rule* set that allows identifying individually all the rules that were used when building the path condition's typed graph.

**Notation** As for match-apply models and transformation rules, we will often abbreviate a path condition  $pc = \langle V, E, st, \tau, Match, Apply, Rule \rangle \in \text{PATHCOND}_{tg}^{sr}$  to its typed graph part  $\langle V, E, st, \tau \rangle$ . We will also refer to the set of transformation rules in  $pc$  identified by the *Rule* relation as  $Rule(pc)$ . Finally, because a path condition is essentially a typed graph, we extend the basic notation of operators and homomorphisms on typed graphs defined in Section 3 to path conditions.

### 4.3 Path Condition Generation Algorithm

The rule combination approach described above was to provide an intuition of what path conditions represent. This section will describe how path conditions are constructed for a transformation using our approach.

Figure 8a outlines the path condition generation algorithm. The algorithm will examine each transformation layer in turn. Path conditions from the previous layer will be combined with rules from the current layer to create a new set of path conditions. This new set of path conditions will then be combined with the rules from the next layer to produce yet another set of path conditions, and so on. At the end of the algorithm, a complete set of path conditions for the entire transformation will be produced.

We now define what is occurring in the 'combination step' in Figure 8a. This step begins by selecting each path condition in the working set, one at a time. Note that at the

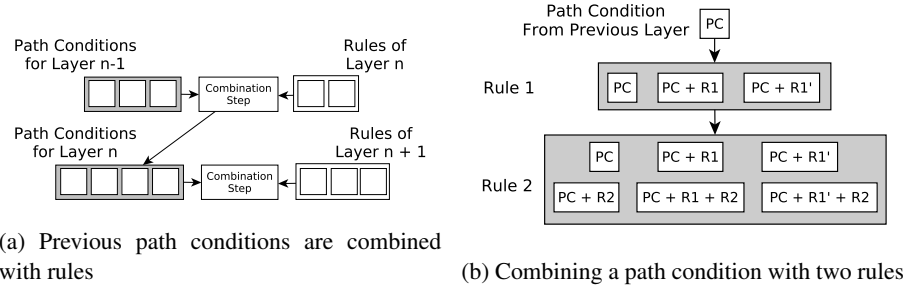


Fig. 8: Two components in the path condition creation process

beginning of the path condition creation process, this working set consists of an empty path condition.

A new set of path conditions will then be created by sequentially combining each rule in the layer with the path condition selected. Recall that a path condition represents a set of rules that have executed, thereby defining a set of transformation executions. Combining a path condition with a rule will produce one or more path conditions which abstract over whether that rule executed on the input model or not. These different path conditions will represent the various pre- and post- conditions implied by the execution of that rule. This is represented in Figure 8b, where a path condition is combined with two rules.

All path conditions produced by the combination steps in the current layer will then be collected to produce the final working set of path conditions for the layer, as shown in Figure 9.

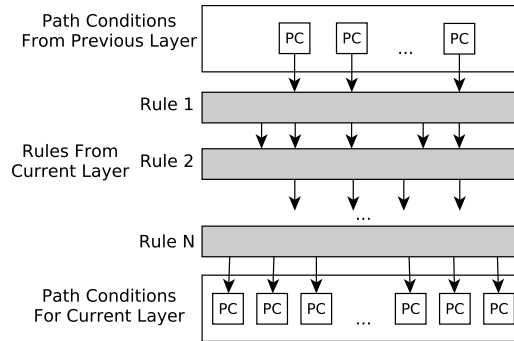


Fig. 9: Creating all path conditions for a layer

#### 4.4 Combining a Path Conditions with a Rule

We will now examine the combination step between one path condition and one rule to produce a set of new path conditions. Let PC be the path condition selected from layer n-1, and R the rule selected from layer n. When PC and R are combined, there are four possibilities based on the dependencies between PC and R:

1. R has **no** dependencies
2. R has dependencies and **cannot** execute
3. R has dependencies and **may** execute
4. R has dependencies and **will** execute

These dependencies are defined by the backward links within R. As mentioned in Section 4.2, backward links enforce that the elements in the apply graph were created by the connected elements in the match graph. In the context of combining a rule and a path condition, these backward links define dependencies between the rule and the elements created by the rules represented by the path condition.

**Definition 16.** *Combination of a Path Condition with a Rule*

Let  $pc = \langle V', E', st', \tau', Match', Apply', Rule' \rangle \in \text{PATHCOND}_{ig}^{sr}$  be a path condition and  $rl = \langle V'', E'', st'', \tau'', Match'', Apply'' \rangle \in \text{RULE}_{ig}^{sr}$  be a transformation rule, where their respective typed graphs are potentially joint. The union of  $pc$  with  $rl$  is built using the operator  $\sqcup^{trace} : \text{PATHCOND}_{ig}^{sr} \times \text{RULE}_{ig}^{sr} \rightarrow \text{PATHCOND}_{ig}^{sr}$ , as follows:

$$pc \sqcup^{trace} rl = \langle V, E, st, \tau, Match, Apply, Rule \rangle$$

where we have that  $V = V' \cup V''$ ,  $E' \cup E'' \subseteq E$ ,  $st' \cup st'' \subseteq st$ ,  $\tau' \cup \tau'' \subseteq \tau$  and if  $v_1 \xrightarrow{e} v_2 \in E \setminus E' \cup E''$  then we have that  $v_1 \in \text{Apply}(V'')$ ,  $v_1 \notin \text{Apply}(V')$ ,  $v_2 \in \text{Match}(V'')$  and also that  $\tau'(e) = \text{trace}$ . Additionally,  $Match$  and  $Apply$  are the typed graph union of the  $Match$  and  $Apply$  graphs of  $pc$  and  $rl$  and  $Rule = Rule' \cup rl$ .

When a path condition is combined with a rule their typed graphs are united. Additionally, symbolic traceability links are built between the newly added apply elements of the rule and all of the rule's match elements. Note that the fact that the graphs are potentially joint allows us to overlap a rule with the path condition by anchoring the rule on traceability link glue in the path condition.

The below figures will demonstrate the four cases above. As a reminder of visual notation, backward links are dashed lines between the match and apply graphs of the rule and path condition, while traceability links are solid lines between the two graphs.

**No Dependencies** The rule R has a match graph which represents its pre-conditions. For a particular transformation execution, it is possible that this match graph would not match the input graph, and thus R would not execute in this transformation executions. To represent all such transformation executions where the rule R would not execute, PC is copied unchanged to the new set of path conditions.

To represent the transformation executions where the match graph of R would match, and therefore R would execute, a new path condition is produced which consists of the union between R and PC. This situation is seen in Figure 10 and formally defined in Definition 17.

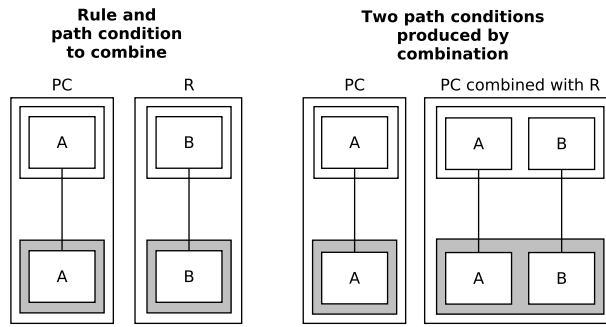


Fig. 10: R has no dependencies

**Definition 17.** *Path Condition and Rule Combination – No Dependencies*

The combination of a path condition and a rule relation  $\xrightarrow{\text{combine}} \subseteq \text{PATHCOND}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr}) \times \text{RULE}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr})$  when the rule has no dependencies is defined as follows:

$$\frac{rl = \langle V, E, st, \tau, Match, Apply \rangle, \nexists e \in E. \tau'(e) = \text{backward}}{\langle pc, PC, rl \rangle \xrightarrow{\text{combine}} PC \cup \bigcup_{pc \in PC} pc \sqcup rl}$$

**Resolving Dependencies** If R contains backward links and thus R defines dependencies on PC, then we need to analyse whether PC can satisfy those dependencies. This is done by matching the backward links in R over the traceability links in PC. Note that traceability links in R are not required to be found in PC, and that only backward links define dependencies.

*Unsatisfied Dependencies* If the backward links in R cannot be matched to traceability links in PC, then R cannot execute after the rules represented by PC. Again, PC will be copied unchanged to the new set of path conditions. This case is shown in Figure 11,

where the backward links between the two B elements in R cannot match over the traceability link in PC. Definition 18 describes this case formally.

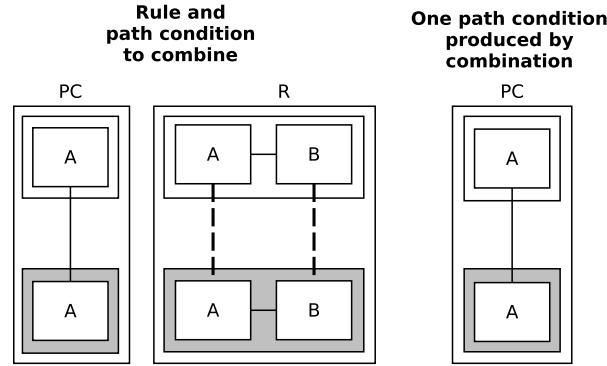


Fig. 11: R's dependencies are not satisfied by PC

**Definition 18.** *Path Condition and Rule Combination – Unsatisfied Dependencies*

The combination of a path condition and a rule relation  $\xrightarrow{\text{combine}} \subseteq \text{PATHCOND}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr}) \times \text{RULE}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr})$  when a rule has dependencies that are not satisfied by the path condition is defined as follows:

$$\frac{\neg(rl|_{\text{backward}} \blacktriangleleft pc|_{\text{backward}})}{\langle pc, PC, rl \rangle \xrightarrow{\text{combine}} PC}$$

**Levi** ▶ **A path condition A does not satisfy the dependencies of a path condition B if there is no injective typed graph homomorphism between the backward links of B and those of A. The injective homomorphism is necessary (not an isomorphism) because it may be that there are more than two of the same backward links in B. Note that there also may be more than two of the same backward links in A, but that problem is covered by the fact that the homomorphism is not surjective.** ◀

*Partially- and Totally- Satisfied Dependencies* Now consider the case where the backward links of R can be found in PC, and R's dependencies are met. The issue then becomes whether R **will** always execute, or whether R **may** execute.

The match graph of R, along with R's backward links, is matched to PC's match graph and traceability links. If all of these elements are found, then we denote this as the 'totally-satisfied case', and R will execute. Otherwise, it is the 'partially-satisfied' case,

and R may execute. Note that we break up these cases for explanation only. Formally, both cases are encompassed by Definition 20.

In the totally-satisfied case, R will be “glued” overtop PC, as seen in Figure 12a. This gluing operation is anchored where the backwards links in R match over the traceability links in PC. The purpose of this operation is to include any elements in R’s apply graph that may not exist in PC. Thus, all elements and associations which exist in both PC and R are ignored. Note that if multiple total matches exist in PC, that R will be glued at multiple points as seen in Figure 12b.

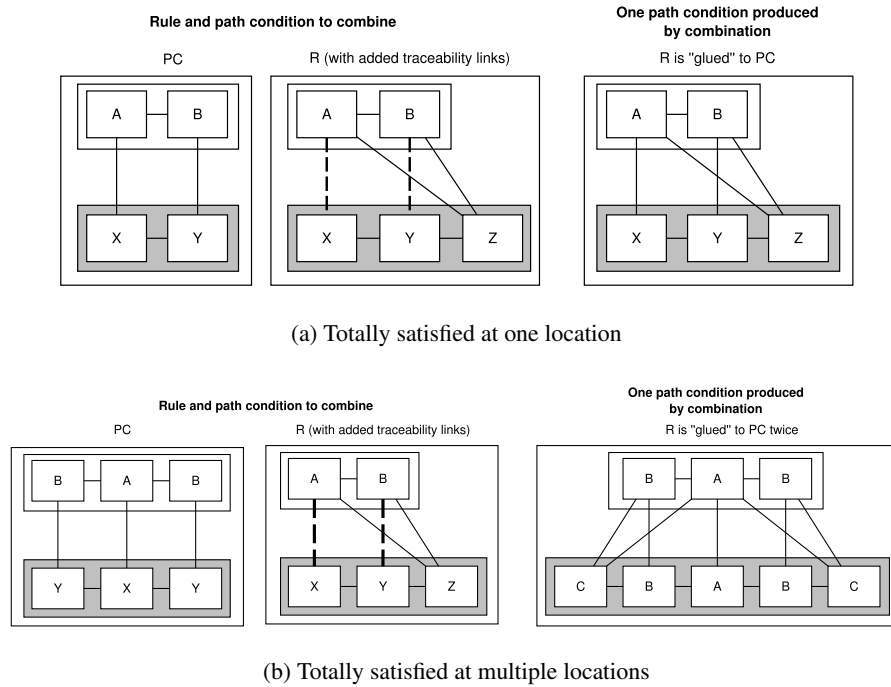


Fig. 12: R’s dependencies are totally satisfied by PC

In the partially-satisfied case, rule R may or may not execute. Note that in Figure 13, PC does not have the association between the A and B elements in the match graph. This means that it is possible for the input model for the transformation to not have this association present, in which case R would not execute. Figure 13 shows the two path conditions produced in this case. The first produced is a copy of PC, where R is assumed not to execute. The second is where R is assumed to execute, at this particular location. Therefore, R is glued onto PC, with the gluing step the same as in the totally-satisfied case above.



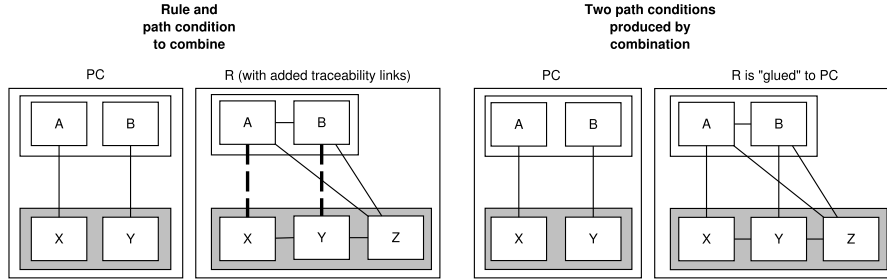


Fig. 13: B's dependencies are partially satisfied by A

Note that this gluing procedure must consider all matching possibilities. For example, in Figure 14, rule R has a backward link that can be partially matched on either the left-hand or right-hand backward links in PC. Therefore, there are four possibilities for how R would match over PC: not at all, on the LHS, on the RHS, or on both sides. These four possibilities define the four new path conditions created.

The first is a copy of PC, as R is assumed to not execute and will produce no new elements. The second is where R will be glued on top of the backward links on the LHS, to add the elements that do not exist in PC already. The third is where the gluing will occur on the RHS. The fourth path condition produced is the case where R will be glued at both locations.

Note that rules may also contain transitive links in their match graphs. In this case, the partial or total matching of R onto PC must consider all transitive matches in order to produce all valid path conditions.

**Definition 19.** *Partial and Total Combinations of a Set of Path Conditions with a Rule Having Dependencies*

The rule combination step relation  $\xrightarrow{\text{rulestep}} \subseteq \text{PATHCOND}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr}) \times \text{RULE}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr})$  is defined as follows: **Levi** ▶ the types of the traceability links in the rules are not correct, they are now 'back' and they need to be 'trace' such that the gluing works ◀

$$\frac{rl \cong rl_{glue} \sqcup ma_{\Delta}}{\langle PC, rl, rl_{glue} \rangle \xrightarrow{p\_comb} PC \cup \bigcup_{pc \in PC} pc \sqcup ma_{\Delta}} \quad \text{trace}$$

$$\frac{rl \cong rl_{glue} \sqcup ma_{\Delta}}{\langle PC, rl, rl_{glue} \rangle \xrightarrow{t\_comb} \bigcup_{pc \in PC} pc \sqcup ma_{\Delta}} \quad \text{trace}$$

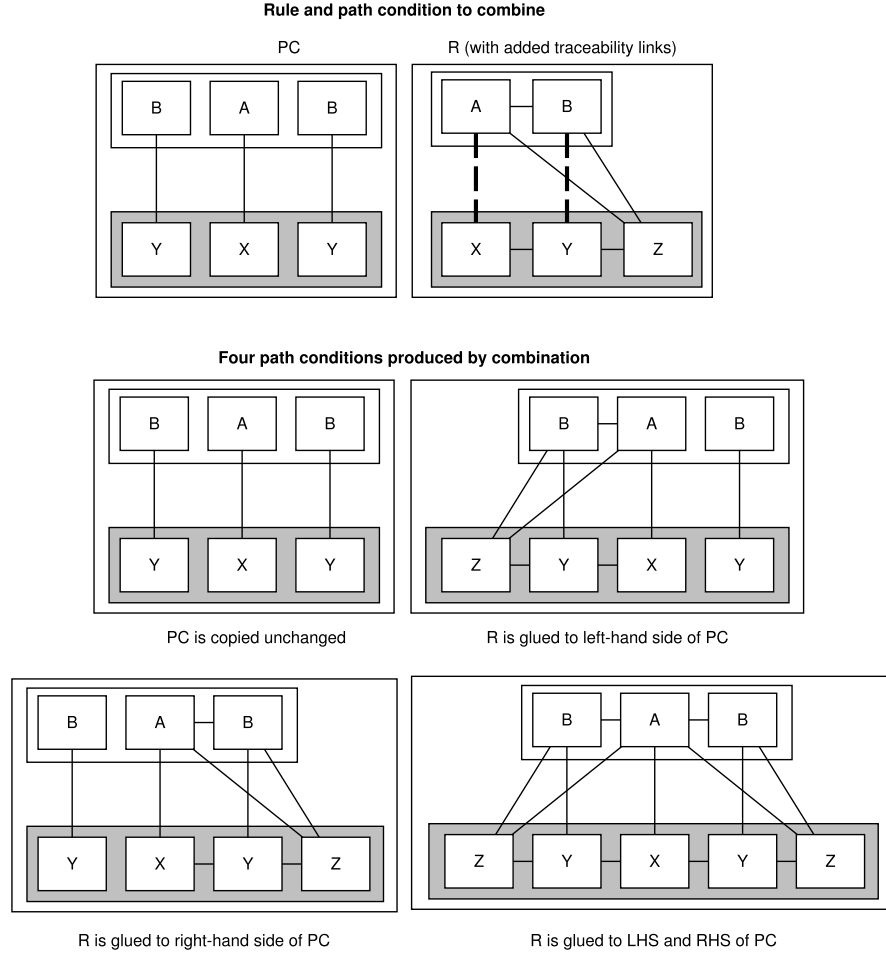


Fig. 14: R's dependencies are partially satisfied by PC, and are glued at all possible matches

$$\begin{array}{c}
 \overline{\langle PC, \emptyset \rangle \xrightarrow{p\_step} PC} \\
 \\
 \frac{
 \begin{array}{c}
 rl_{glue} \in partialSet, \langle PC, rl, rl_{glue} \rangle \xrightarrow{p\_comb} PC'', \\
 \langle PC'', rl, partialSet \setminus \{rl_{glue}\} \rangle \xrightarrow{p\_step} PC'
 \end{array}
 }{
 \langle PC, rl, partialSet \rangle \xrightarrow{p\_step} PC'
 } \\
 \\
 \overline{\langle PC, \emptyset \rangle \xrightarrow{t\_step} PC}
 \end{array}$$

$$\frac{rl_{glue} \in totalSet, \langle PC, rl, rl_{glue} \rangle \xrightarrow{t\_comb} PC'', \quad \langle PC'', rl, totalSet \setminus \{rl_{glue}\} \rangle \xrightarrow{t\_step} PC'}{\langle PC, rl, totalSet \rangle \xrightarrow{t\_step} PC''}$$

When a rule is combined with a path condition, partial and total matches of the rule may exist in the path condition. A total match of the rule is found in the path condition if a surjective typed graph homomorphism exists between the backward matcher of the rule and a subgraph of the path condition ( $rl_{glue} \sqsubseteq pc \wedge rl \blacktriangleleft rl_{glue}$ ). As a reminder, the backward matcher of a rule consists of the rule's match part, together with its backward links (see Definition 36). In this case, existing path conditions in the output set will be combined with... These two cases are modeled by the two inference rules in Definition 19. For each case a delta is added to the glue graph found such that the necessary apply or match part of the rules are added to the original path condition. In case the match is partial, then the previous set of path conditions is kept where the delta has been added, as well a second possibility with the total rule. Note that the fact that we use a surjective homomorphism allows us to deal with both rules that are isomorphic or homomorphic to parts of PC. This allows us to have more backward links in the path condition than in a rule, but also the other way around.

**Definition 20.** *Path Condition and Rule Combination – Partially and Totally Satisfied Dependencies* The combination of a path condition and a rule relation  $\xrightarrow{combine} \subseteq \text{PATHCOND}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr}) \times \text{RULE}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr})$  when the rule has dependencies that are satisfied by the path condition is defined as follows:

$$\frac{rl|_{backward} \blacktriangleleft PC|_{backward}, \quad \langle PC, rl, partialsat(rl^{backM}, pc) \rangle \xrightarrow{p\_comb} PC'', \quad \langle PC'', rl, totalsat(rl^{backM}, pc) \rangle \xrightarrow{t\_comb} PC'}{\langle pc, PC, rl \rangle \xrightarrow{combine} PC'}$$

where

$$rl_{glue} \in partialsat(rl, pc) \Leftrightarrow rl_{glue} \sqsubseteq pc \wedge rl|_{backward} \blacktriangleleft rl_{glue} \wedge \nexists rl'. (rl_{glue} \sqsubseteq rl' \sqsubseteq pc \wedge rl^{backM} \blacktriangleleft rl')$$

and

$$rl_{glue} \in totalsat(rl, pc) \Leftrightarrow rl_{glue} \sqsubseteq pc \wedge rl^{backM} \blacktriangleleft rl_{glue}$$

**Levi** ► **CAREFUL WITH THE SURJECTIVE HOMOMORPHISM IN THE MATCHING!!!** *The combstep relation starts with the first path condition in PC and checks each rule in the path condition to combine for partial or total matches. The multiplied path conditions resulting from the partial or total matches are added to PC along until no more rules exist. Note that this does not induce a problem with parallelism in rule application because either the rule is isomorphic to a part of the path condition, in which case backward links are not touched, or there is a surjective homomorphism between the rule and a part of the original path condition, in which case a new backward link can be added that can be used by other rules. That is fine because if more backward links exist on the second path condition that means that an assumption was made on the number of elements consumed by similar previous rules. The first argument of the relation holds the original path condition such subgraphs to be glued can be fetched inside. We assume that when we create path conditions their nodes and edges are not disjoint.* ◀

**Considering Further Rules** Thus far the algorithm has created a set of path conditions that represent how one rule from a layer will add new elements to one path condition from the previous layer. These path conditions are then themselves combined with the next rule in the layer in the same manner.

Note that the choice of next rule does not matter, due to the rule non-interference guaranteed by the semantics of DSLTrans. In order to represent this non-interference in the construction of path conditions, we specify that the matching of rule dependencies is against the path condition from the previous layer, not the specific path condition the rule is to be combined with. This ensures that the result of combining one rule with a path condition will have no impact on how following rules will combine.

The combination of one path condition with all the rules in the layer will produce a new set of path conditions. After this step is repeated for all the path conditions in the previous layer, these new sets of path conditions are collected together to produce the working set of path conditions for the layer. This is depicted in Figure 9, and formalized in Definition 21.

This working set is then itself combined with the rules in the next layer as in the algorithm just described, to obtain yet another working set of path conditions. This process will then continue in this layer-by-layer fashion through the transformation.

After all layers have been processed, the working set of the last layer contains all the possible path conditions of the transformation. Definition 23 formally describes this process. Through our abstraction relation, the created path conditions will represent ev-

ery feasible transformation execution. Section 6 will discuss how our algorithm proves properties on these path conditions, and thus on all executions of the transformation.

**Definition 21.** *Combining a Path Condition with a Layer*

The path condition rule step relation  $\xrightarrow{\text{rulecomb}} \subseteq \text{PATHCOND}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr}) \times \text{RULE}_{tg}^{sr} \times \text{PATHCOND}_{tg}^{sr}$  relation is defined as follows:

$$\frac{}{\langle pc, PC, \emptyset \rangle \xrightarrow{\text{combpclyer}} PC}$$

$$\frac{rl \in layer, \langle pc, PC, rl \rangle \xrightarrow{\text{combine}} PC'', \langle pc, PC'', layer \setminus \{rl\} \rangle \xrightarrow{\text{combpclyer}} PC'}{\langle pc, PC, layer \rangle \xrightarrow{\text{combpclyer}} PC''}$$

**Definition 22.** *Combining a Set of Path Condition with a Layer*

The path condition layer step relation  $\xrightarrow{\text{combpcsetlayer}} \subseteq \text{PATHCOND}_{tg}^{sr} \times \text{TRANSF}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr})$  relation is defined as follows:

$$\frac{}{\langle \emptyset, layer \rangle \xrightarrow{\text{combpcsetlayer}} \emptyset}$$

$$\frac{pc \in PC, \langle pc, \{pc\}, layer \rangle \xrightarrow{\text{combpclyer}} PC', \langle PC \setminus \{pc\}, layer \rangle \xrightarrow{\text{combpcsetlayer}} PC''}{\langle PC, layer \rangle \xrightarrow{\text{combpcsetlayer}} PC' \cup PC''}$$

**Definition 23.** *Path Condition Generation*

Let  $[layer :: tr] \in \text{TRANSF}_{tg}^{sr}$  be a transformation, where  $layer \in \text{LAYER}_{tg}^{sr}$  is a Layer and  $tr$  also a transformation. The  $\xrightarrow{\text{pathcondgen}} \subseteq \mathcal{P}(\text{PATHCOND}_{tg}^{sr}) \times \text{LAYER}_{tg}^{sr} \times \text{PATHCOND}_{tg}^{sr}$  is defined as follows:

$$\frac{}{\langle PC, [] \rangle \xrightarrow{\text{pathcondgen}} PC}$$

$$\frac{\langle \epsilon_{pc}, \{\epsilon_{pc}\}, layer \rangle \xrightarrow{\text{layercomb}} PC'', \langle PC'', tr \rangle \xrightarrow{\text{pathcondgen}} PC'}{\langle \epsilon_{pc}, [layer :: tr] \rangle \xrightarrow{\text{pathcondgen}} PC}$$

## 5 Abstraction Relation between Path Conditions and Transformation Executions

In this section we argue that our path condition building algorithm is both *valid* and *complete*. In this context *validity* means that for each path condition there exists at least one transformation execution that it abstracts. In other words, no path conditions are produced that lack a concrete transformation execution counterpart. *Completeness* of the symbolic execution means that every transformation execution is abstracted by at least one path condition.

In order to do this we need to define the abstraction relation between the execution of a DSLTrans transformation and the path condition that represents it. This abstraction relation allows us to manipulate a finite set of representative path conditions during property proof and thus guarantees the decidability of our technique. As well, it enforces the fact that the match part of the path condition can be injectively found in the execution's input model, and that all components of the apply part of the path condition, including traceability information, can be found in the output model of the execution at least once (given they are necessarily produced).

### Definition 24. Abstraction of a Transformation Execution by a Path Condition

Let  $tr \in \text{TRANSF}_{ig}^{st}$  be a DSLTrans transformation. Let also  $pc = \langle V_{pc}, E_{pc}, st_{pc}, \tau_{pc}, Match_{pc}, Apply_{pc}, Rules_{pc} \rangle \in \text{PATHCOND}(tr)$  of be a path condition of  $tr$  and  $ex = \langle V_{ex}, E_{ex}, st_{ex}, \tau_{ex}, Input_{ex}, Apply_{ex} \rangle \in \text{Exec}(tr)$  be an execution of  $tr$ . We have that  $ex$  is abstracted by  $pc$ , noted  $ex \Vdash pc$ , if and only if:

$$(\forall rl \in Rules_{pc}. Match(rl) \triangleleft Input_{ex}^*) \wedge ex^{traceOut} \blacktriangleleft pc^{traceApply}$$

where we have that:  $ex^{traceOut} = \langle V'_{ex}, E_{ex} \setminus Input(E_{ex}), st'_{ex}, \tau'_{ex} \rangle \sqsubseteq ex$  is the output part of  $ex$ , including traceability links;  $pc^{traceApply} = \langle V'_{pc}, E_{pc} \setminus Match(E_{pc}), st'_{pc}, \tau'_{pc} \rangle \sqsubseteq pc$  is the output part of  $pc$ , including symbolic traceability links.

### 5.1 Examples

In this section, we provide a number of examples to demonstrate the abstraction relation.

**Empty Path Condition** We begin by defining which transformation executions an empty path condition will abstract. Figure 15 demonstrates two cases. In each, the path condition is on the left-hand side, and a transformation execution is on the right-hand

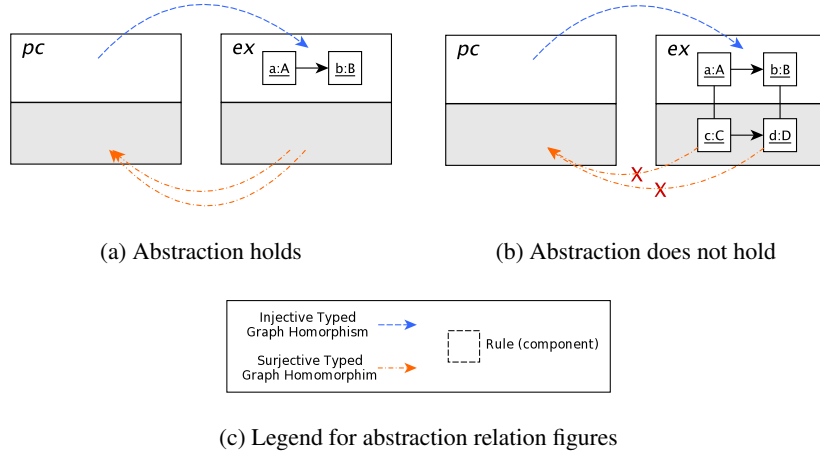


Fig. 15: Abstraction of transformation executions by the empty path condition

side. Note that in Figure 15a, the path condition abstracts the transformation execution, while in Figure 15b, the abstraction relation does not hold.

The match part of the path condition represents the pre-conditions for the path condition to be true, depending on which rules have symbolically executed in the transformation. For example, if the match graph is empty, this represents all executions where no rules have executed.

The first condition for the abstraction relation is to determine whether an injective match can be made between the match graph of the path condition, and an transformation executions. Note that in both Figure 15a and Figure 15b, an empty injective match can be made, highlighted by blue arrows.

The second condition for the abstraction relation is whether a subjective match can be made from the transformation execution's output model to the apply graph of the path condition. This is represented by orange arrows in Figure 15a and Figure 15b. The match is surjective as there may not be any elements in the output model that are not represented by the path condition's apply graph. Note that multiple elements in the output model may match to the same element in the apply graph of the path condition. This is expected, as the structure found in the apply graph may be found multiple times in the output model.

The empty apply graph of the path condition defines no post-conditions on the output model, as no rules have executed. Note that there is the empty surjective match between the output model of the transformation execution in Figure 15a and the path condition. This is intuitive, as the lack of elements in the output model means no rules

have executed, which corresponds to the lack of post-conditions defined by the path condition.

In contrast, there is no surjective match between the elements of the output model in Figure 15b and the path condition. Note that the transformation execution has elements in the output model and thus at least one rule must have executed. However, the path condition does not represent that a rule has executed. Therefore, the path condition shown cannot represent this execution. The fact that this transformation execution will be covered by a path condition is shown by Proposition 8.

**Non-overlapping Rule Components** This second example shows the abstraction relation when the path condition represents the symbolic execution of a number of rules. In addition, no rules share elements in these examples.

Let us first examine how the injective match operates between the elements in the path condition and the transformation executions in Figure 16a and Figure 16b. Note that this injective match can be found in both cases.

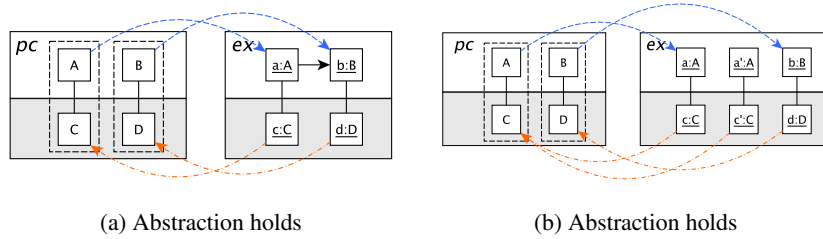


Fig. 16: Abstraction of transformation executions by non-overlapping rule components

Similarly, there is a surjective match between the elements of the output model for both transformation execution and the apply graph of the respective path condition. Note that this surjective match also holds in Figure 16b, where examination of the transformation execution shows that one rule has executed twice. The abstraction relation abstracts over the number of times that a rule has executed.

We also note that these matches must also match over associations between the elements, including association typing. This is not included in the figures for visual clarity.

**Overlapping Rule Components** For these examples, the path conditions contain overlapping rule components. This is to represent the interaction of rule elements, where the



elements may match over the same or different elements in the transformation execution.

For example, the two components in Figure 17a correctly match over the transformation execution shown. The abstraction relation holds due to the injective match, which allows the match elements from different components to match to the same input model element.

In contrast, Figure 17b shows an example where the abstraction relation does not hold. Note that a component in the match graph of the path condition contains two B elements. Both of these elements must be found in the transformation execution, and thus it is not correct for them to injectively match to the same element in the input model.

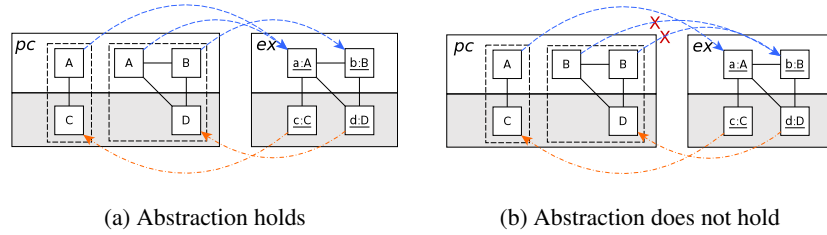


Fig. 17: Abstraction of transformation executions by overlapping rule components

**Indirect Links** As a final example, we present a path condition that includes indirect links. For this injective match to hold, the elements at both ends of the link must be found, and there must be an indirect link between the matched elements. Note that only the match graph of the path condition may include an indirect link.

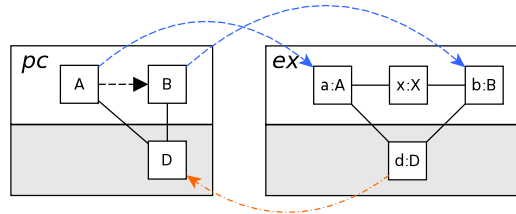


Fig. 18: Abstraction holds with indirect links

## 5.2 Validity and Completeness

**Proposition 1.** (*Validity*) *Every path condition abstracts at least one transformation execution.*

*Proof sketch.* blah

□

**Proposition 2.** (*Completeness*) *Every transformation execution is abstracted by one path condition.*

*Proof sketch.* Our path conditions construction algorithm takes all possible interactions of rules into account:

1. When building the path conditions for one layer we have built all the combinations of rules, as well as their possible interactions through *disambiguation*;
2. When combining the path conditions from different layer different layers we have considered the cases where no interactions exist, or where backward links in rules requires certain elements to already exist in the generated model thus far.

Note that, given the considered subset of DSLTrans constructs as described in Section 2, no additional cases of rule interaction other than the ones described above need to be considered. From Proposition 7 we also know that at least one concrete transformation execution exists per path condition. In the proof of Proposition 7 we have built, for each path condition  $A$ , the simplest transformation execution that is abstracted by  $A$ . These transformation executions produce one instance of each element and association present in the apply pattern of  $A$ . However, given ?? in ?? (abstraction of a transformation execution by a path condition), an infinite amount of transformation executions are always abstracted by  $A$ . These executions correspond to the generation of output models holding an arbitrarily large amount of the elements and associations while still being abstracted by  $A$ . Because we have considered all possibilities of execution in our path condition construction algorithm, the union of all transformation executions built for each path condition constitutes the complete, infinite set of transformation executions. We thus know that every possible transformation execution is abstracted by at least one path condition.

□

**Lemma 1.** *A Transformation Execution is Abstracted by Exactly One Path Condition*

*Proof sketch.* blah

□

## 6 Verifying Properties of DSLTrans Transformations

The algorithm presented in Section 4 will produce all possible path conditions for a given DSLTrans transformation. This section will detail our second contribution: a method to prove properties on all of these path conditions, and thus on the transformation itself. This relies on an abstraction relation allows us to prove properties on each path condition in the final path condition set and have that result hold on all transformation executions abstracted by that path condition. This is formally stated in ??.

The properties we are interested in have an implication form. Similarly to rules and path conditions, properties are largely composed of two patterns. They represent the following statement: if this pattern is found in the input model, then this other pattern must be found in the output model, possibly including traceability constraints. As such, we encode properties as a set of pre-condition and post-condition patterns.

Pre-conditions use the same pattern language as the match graph in DSLTrans rules, allowing the possibility of including several occurrences of the same metamodel element as well as indirect links in the property. Indirect links in properties have the same meaning as in the rule match graph – they involve patterns over the transitive closure of containment links in input models.

Post-conditions also use the same patterns language as the apply graph 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. Traceability links can also be used in properties to impose traceability relations between pre-condition and post-condition elements. A formal definition of our property language can be found in Definition 25.

In figures 19a and 19b we present two properties we wish to prove or disprove regarding all executions of the transformation presented in Figure 2. The property in Figure 19a represents the statement “*a model which includes a police station that has both male and female officers will be transformed into a model where the male officer will exist in the male set and the female officer will exist in the female set*”. This is something we expect will always hold in our transformation. The property in Figure 19b represents the statement “*any model which includes a 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.

We prove these properties on each path condition created by checking whether the pre-condition graph of the property can be isomorphically matched onto the match graph of the path condition, as in Definition 27. If this match cannot be performed,

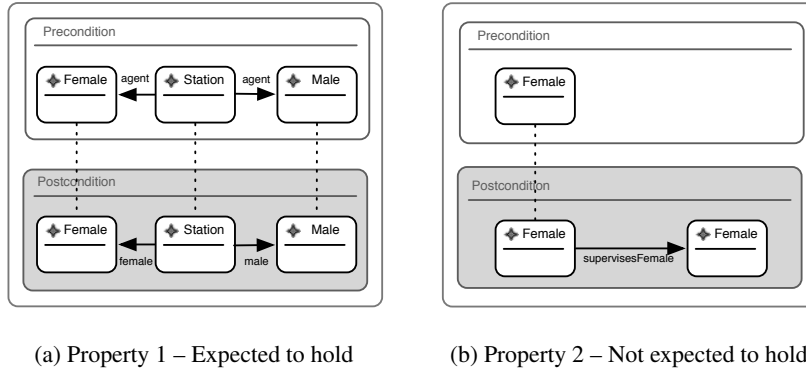


Fig. 19: Properties to be proved on the Police Station transformation

then the pre-condition does not hold for this path condition and this property will not be checked for this property.

If the pre-condition can be isomorphically matched to the match graph of the path condition, then we check if the entire property can be isomorphically matched to the path condition. Again, this is described in Definition 27. If this matching cannot be performed, then the property does not hold. The path condition itself then serves as a counter-example for the transformation property. The path condition also contains information on which rules were executed in the counter-example to permit further examination.

This property-checking can be done relatively quickly with efficient graph-matching techniques. Section 7 will briefly describe implementation and optimization details, while Section 8 presents two property-proving experiments.

As for the path condition building algorithm, *validity* and *completeness* need to be examined regarding our property verification algorithm. In this context *validity* means that there exists an equivalence between the result of verifying a concrete model transformation execution for a particular input model, and the result of verifying the path condition which abstracts that concrete transformation execution. This equivalence will rely on the abstraction relation formally presented in ??.

**Definition 25.** *Property of a Transformation*

Let  $t$  be a DSLTrans transformation having source metamodel  $s$  and target metamodel  $t$ . A property of  $t$  is a 7-tuple  $\langle V, E \cup Il, \tau, Match, Apply, Bl, Il \rangle$ , where  $\langle V, E, \tau, Match, Apply \rangle \in MAP_t^s$  is a match-apply pattern.  $Match = \langle V', E', \tau', s \rangle$ ,  $Apply = \langle V'', E'', \tau'', t \rangle$ , the edges  $Tl \subseteq V' \times V''$  are called backward links 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  $Property_t^s$ .

It is important to mention the fact that the property language, as defined in Definition 25, is based solely on the source and target metamodels of a DSLTrans model transformation. This property language is an over-approximation of the property language that our algorithm practically verifies. This over-approximation is due to the fact that certain properties that can be expressed may refer to patterns that are never matched by the transformation. In one of our previous articles on this topic [4] we referred to these properties as being *non-provable*. For our theoretical purposes we assume we only deal with *provable* properties: properties for which the pre-condition appears in the match pattern of at least one path condition of the transformation being verified.

Definition 26 details how a transformation execution is said to satisfy a property. Due to the common structure between properties and transformation executions, this satisfaction is based on whether the property can be isomorphically matched in the transformation execution.

**Definition 26.** *Satisfaction of a Property by an Execution of a Transformation*

Let  $t$  be a DSLTrans transformation having source metamodel  $s$  and target metamodel  $t$ . Let also  $p = \langle V_p, E_p, \tau_p, Match_p, Apply_p, Bl_p, Il_p \rangle \in Property_t^s$  be a property of  $t$  and  $ex = \langle V_x, E_x, \tau_x, Match_x, Apply_x, Tl_x \rangle \in Exec_t^s$  be an execution of  $t$ . Execution  $ex$  satisfies property  $p$ , written  $ex \models p$ , if and only if:

$$Match_p \triangleleft Match_x^* \implies p \triangleleft ex^*$$

More informally, **if** there exists a typed graph injective homomorphism between  $Match_p$  and  $Match_x^*$  **then** there exists a typed graph injective homomorphism between  $p$  and  $ex^*$ .

Note that in Definition 26 we use the typed graph injective homomorphism relation ' $\triangleleft$ ' between patterns, which are structures that contain more information than the typed graphs over which the relation ' $\triangleleft$ ' was originally defined. In the text that follows we will often make use of such notational abuse with the aim of simplifying our presentation.

Definition 27 is similar to that of property satisfaction for a transformation execution, but for path conditions. Again, the property is isomorphically matched onto the respective graphs in the path condition.

**Definition 27.** *Satisfaction of a Property by a Path Condition*

Let  $t$  be a transformation,  $p = \langle V_p, E_p, \tau_p, Match_p, Apply_p, Bl, Il_p \rangle \in Property_t^s$  be a property of  $T$  and  $pc = \langle V_{pc}, E_{pc}, \tau_{pc}, Match_{pc}, Apply_{pc}, Tl, Il_{pc} \rangle \in PC_t^s$  be a path condition of  $t$ . Path condition  $pc$  satisfies property  $p$ , written  $pc \vdash p$ , if and only if:

$$Match_p \triangleleft Match_{pc}^* \implies p \triangleleft pc^*$$

More informally, **if** there exists a typed graph injective homomorphism between  $Match_p$  and  $Match_{pc}^*$  **then** there exists a typed graph injective homomorphism between  $p$  and  $pc^*$ .

Given these definitions, we see that as all transformation executions are represented by path conditions, proving properties on the path conditions allows us to conclude property validity on all transformation executions.

**Validity** In this section we prove the validity of our property proof algorithm by considering what it formally means when a property is proved on a path condition. We argue that if the property holds or not on the path condition, then it will accordingly hold or not on for any transformation executions that the path condition represents.

**Proposition 3.** (Validity) *The result of checking a property on a path condition and of checking the property on all the transformation executions that path condition abstracts will be the same.*

Let  $t$  be a DSLTrans transformation having source metamodel  $s$  and target metamodel  $t$ . Let also  $ex = \langle V_x, E_x, \tau_x, Match_x, Apply_x, Tl_x \rangle \in Exec_t^s$  be a transformation execution of a transformation  $t$ ,  $pc = \langle V_{pc}, E_{pc}, \tau_{pc}, Match_{pc}, Apply_{pc}, Tl_{pc}, Il_{pc} \rangle \in PC_t^s$  be a path condition of transformation  $t$  and  $p = \langle V_p, E_p, \tau_p, Match_p, Apply_p, Bl_p, Il_p \rangle \in Property_t^s$  be a property of  $t$ . This given, we have that:

$$ex \Vdash pc \wedge pc \vdash p \iff ex \models p$$

Proof sketch. blah

□

### Completeness

**Proposition 4.** (Completeness) *Proving that a property holds or does not hold by examining all path conditions of the transformation is equivalent to proving the property holds or does not hold for all transformation executions.*

*Proof sketch.* This result follows from the combination of the validity and completeness of the path condition algorithm, as seen in propositions 7 and 8, as well as the validity of the property proving step on each path condition, as seen in Proposition 9.  $\square$

## 7 Implementation Details

In this section we will briefly describe our implementation. In particular, we highlight optimizations made and provide results suggesting that our algorithms can feasibly scale to industrial-sized applications.

### 7.1 Enabling Technology and Prototype

In previous work we have reported on the usage of Prolog as a means to build a proof-of-concept prototype for our technique [4]. The experiments performed using Prolog were inconclusive regarding the scalability of our technique given that the path condition construction algorithm as now described in Section 4 lacked a formal understanding, as well as several other imprecisions. As such, no performance optimisations were attempted.

Through our sponsorship by the NECSIS (Network for the Engineering of Complex Software-Intensive Systems for Automotive Systems) project, we have the opportunity to apply our verification technique in an industrial setting. In order to achieve high performance in this setting, despite the complexities of verification techniques, we were required to choose an underlying efficient implementation framework. Our goals were to select a framework which: 1) allows graph manipulation natively. This detaches us from the worries of building and optimising our own subgraph isomorphism NP-complete algorithms, which are constantly used during path condition construction; and 2) allows detailed control over graph manipulation such that the implementation of complex optimizations is feasible. These optimizations are potentially required to apply our technique to large and complex model transformations.

We have chosen T-Core [19, 20] as our graph manipulation framework. Aside from satisfying our basic requirements described above, T-Core allows natively rewriting typed graphs, which considerably eases our implementation work. The algorithms described in Section 4 and ?? have been implemented by scheduling T-Core graph manipulation primitives using the Python programming language. The code that was used in the experiments reported in this section can be found at [21].

## 7.2 Complexity

Let us motivate our discussions of optimisation and performance by providing an approximate formula for the complexity of the path condition construction and property proof algorithms presented in sections 4 and 6.

Assume a DSLTrans transformation having  $l$  layers, each of those layers having  $r$  rules. Also, each rule in a layer has  $e$  elements having the same type as elements in other rules of the same layer, which will require disambiguation, and no more than  $c$  pairs of rules per layer share common elements. Assume further that rules belonging to different layers do not require disambiguation. This last condition implies that elements of the input model already matched in a given layer may be referred to in a subsequent layer by using backward links, but that they are not reused for output element creation by a subsequent layer.

Note that this last assumption on the form of a DSLTrans transformation is reasonable and an adequate parametrization of  $l$ ,  $r$ ,  $e$  and  $c$  can over-approximate the form of most DSLTrans transformations we have built up until now. Note that in our calculations below we abstract from the complexity of the graph matching necessary for our algorithms, which in this particular discussion we consider to be a constant operation.

The construction of ambiguous path conditions for one layer requires calculating the power set of the rules in the layer and as such has complexity  $O(2^r)$ . Disambiguating the ambiguous path conditions generated for a layer will then increase that complexity to  $O(2^r + 2^{e \cdot c})$ , given each rule requires  $2^{e \cdot c}$  disambiguations with rules from the same layer. By combining two disambiguated path conditions sets for two layers of the transformation layers the cost is increased to at least  $O((2^r + 2^{e \cdot c})^2)$ , given the cost of the Cartesian product of path condition combination for a pair of layers. This also does not take into account the cost of merging path conditions, which we consider to be roughly linear. However, given the fact that  $l$  layers exist in our transformation, we reach a final complexity for path condition creation:

$$O((2^r + 2^{e \cdot c})^l) \tag{1}$$

Note that, in the worst case where no path conditions belonging to different layers are merged, this number is similar both for time and space complexities.

For our property proving algorithm, the complexity measure is linear in the number of generated path conditions. This is because we are simply matching the property over each path condition, and are considering matching to be constant in this complexity discussion.



### 7.3 Optimisations

In order to tackle the time and space complexities of the path condition construction and property proof algorithms we have employed several engineering strategies. In the following paragraphs we describe the most relevant of these strategies.

- Path condition construction and property proof are very repetitive processes since most individual rules are often composed and searched in the same manner. Since many similar situations have to be investigated during path condition construction and property proof, memoisation was used whenever possible to avoid isomorphic graph matching and rewrite operations. As such caching is heavily used in both algorithms;
- Given the disambiguation algorithm presented in ?? is recursive and presents exponential time complexity, we have performed disambiguation only when strictly necessary: when merging path conditions from different layers (given the disambiguated cases from the previous layer may be required for the merge); and when performing property verification for the path conditions that contain the elements in the pre-condition of the property. Note that the fact that that disambiguation is performed in this ‘lazy’ fashion allows us to operationally keep path conditions as sets of individual rules. This makes it possible to heavily reuse pointers to the original transformation rules when building path conditions, thus reducing the algorithm’s space complexity when compared to the explicit representation of each generated path condition. This also means that, practically, path condition disambiguation is mostly done on demand during property proving;
- We have implemented a method to incrementally build the set of disambiguated path conditions given a set of ambiguous path conditions. The strategy starts by disambiguating the rules that compose the path condition two by two, then disambiguating these results with the remaining rules, and so on. This solution is based on the fact that the disambiguation operation is associative. In this fashion no repeated disambiguated path conditions are built, when compared to a recursive version of the same algorithm. Also, the intermediate results of disambiguating several rules can be cached to be used when disambiguating the same rules in different path conditions;
- For property proof we have implemented a strategy to avoid checking path conditions where the property is sure to hold. The strategy is based on the fact that if a path condition  $B$  contains the same elements as a path condition  $A$  where the property has already been checked successfully, and no additional elements of the property exist in  $B$ , then the property still holds for  $A$ .

## 8 Experiments

This section will detail the experiments we performed in order to measure the performance of our technique. We present timing results for two experiments. The first is to obtain timing results for proving two properties on a synthetic transformation, while the second experiment is sourced from our industrial partners.

### 8.1 Experimental Setup and Results

The complexity of Equation (1) suggests that our property proving approach is intractable in the general case. However, we have provided in Section 7.3 a number of concrete optimisations to allow us to prove properties on transformations of non-trivial size. This section will detail our experiments to determine the effect of the number of rules in the transformation on the performance of our implementation.

For our experiment we have used the Police Station transformation as described in Section 2 as a sample transformation. However, in order to determine the performance characteristics of our approach, we have replicated the rules within the transformation.

This was achieved by synthetically augmenting the original metamodels by replicating their elements twice, thus building source and target metamodels that are three times larger. For example, in the source metamodel we will now have *Station1* (renamed from the original *Station* class) and its replicas *Station2* and *Station3*. These three metamodel elements are distinct from each other and are formally three different types. We have also added new rules that utilise these new types, as seen in Figure 20.

Note that for clarity reasons in Figure 20 we have abbreviated the element names *Station*, *Male* and *Female* to *S*, *M* and *F* respectively. Additionally, the numerical suffix denotes which replicated metamodel element is represented, as described above.

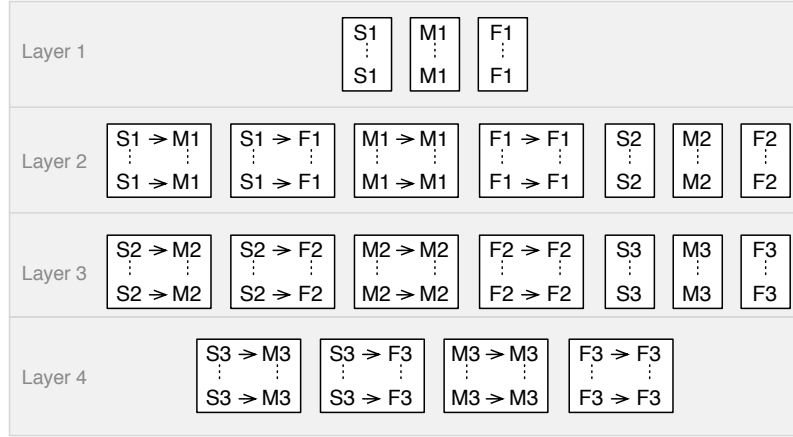


Fig. 20: Replicated Police Station transformation for performance tests

**Results** The results in table 1 were obtained by verifying the the properties in figures 19a and 19b on the transformation seen in Figure 20. The experimental platform was 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 used to run our experiments can be found at [21].

# of rules	# of path conds. created	Path conds. build time (s)	Memory used (KB)	Proof time for property that holds (s)	Proof time for property that does not hold (s)
3	8	<0.01	0.08	-	-
5	16	0.13	0.09	0.19	0.003
7	34	0.39	0.17	1.26	0.003
10	272	1.87	1.24	2.40	0.003
12	442	2.68	1.83	3.40	0.003
14	1156	9.00	4.98	8.38	0.003
17	9248	59.08	38.01	73.51	0.003
19	15028	97.52	60.10	140.77	0.003
21	39304	369.19	156.79	412.02	0.003

Table 1: Results for creating the set of all path conditions and proving two properties

**Time Required to Produce Path Conditions** An important metric for our work is measuring how long it takes to produce the final set of path conditions from a DSLTrans

transformation. As seen in Section 7.2, this metric depends on the composition of the rules in the transformation's layers.

The first column of Table 1 shows the number of rules for each part of the experiment. In order to provide greater granularity in the data, and determine the precise effect of adding more rules to a layer versus adding another layer of rules, we examine subsets of rules taken from Figure 20. For example, the subset with five rules contains the three first rules of layer 1 plus the two first rules of layer 2; the subset with seven rules contains the first three rules of layer 1 plus the four first rules of layer 2; and so on.

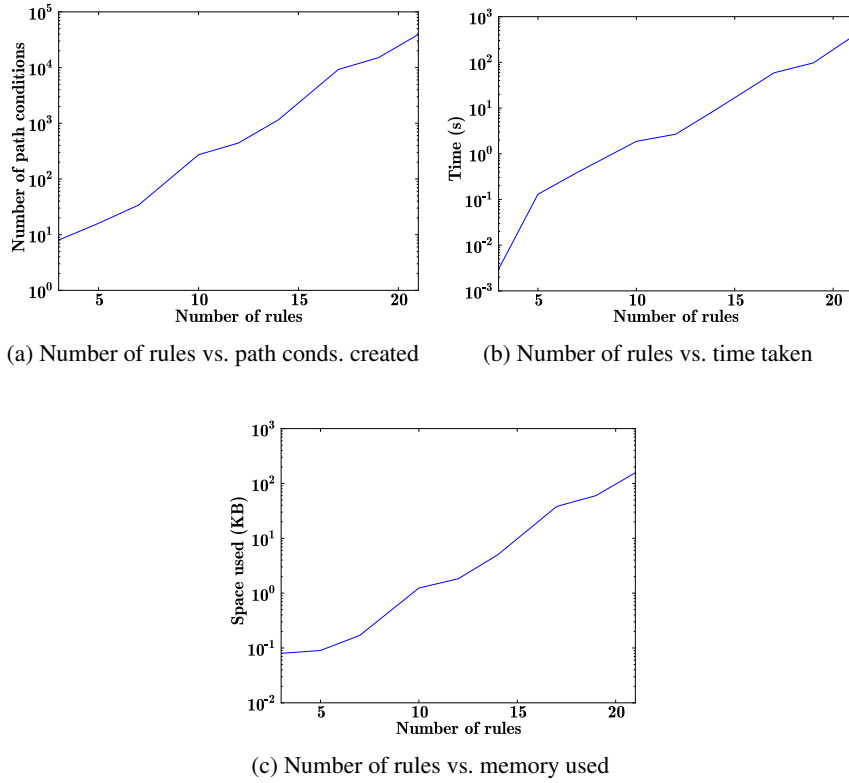


Fig. 21: Metrics for the path condition creation process

Figure 21a presents the number of path conditions created for a given number of rules, while Figure 21b graphs the time taken to create all the path conditions. Both the number of path conditions and the time required to build them rise steeply with the number of rules, but it is quite feasible to build path conditions and prove properties for up to 21 rules. As shown later in the section on industrial experimentation, 21 rules

exceeds the number of rules in our industrial case study. It also exceeds the number of rules in several useful DSLTrans transformations [7–9].

Table 1 and Figure 21c demonstrate that memory consumption is very modest, remaining well under a megabyte for thousands of path conditions. This is due to the optimisations that we perform, such as only storing pointers to path conditions. We are encouraged that this algorithm can scale extremely well in terms of respecting memory constraints.

**Time Required to Prove Properties** We now examine the time it takes to prove two properties on the transformation based on the number of path conditions created from that transformation. The two properties to be proven are shown in Figure 19. The first, in Figure 19a, is a property that we expect to hold for all path conditions. Figure 19b shows a property that we expect to *not* hold for all path conditions.

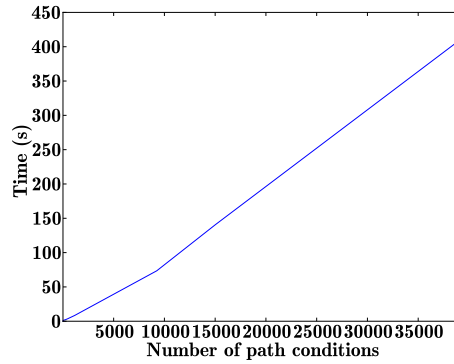


Fig. 22: Time required to prove the property that holds on all path conditions

Figure 22 shows the time in seconds required to prove the property that holds on all path conditions, as seen in the fifth column in Table 1. Note that the time taken increases linearly with the number of path conditions to examine. This increase occurs as each property must be checked to ensure that the property will hold.

In contrast, the time required to disprove the property that does not hold is roughly constant. This can be seen in the sixth column in Table 1, where this proof took 0.003 seconds regardless of the number of path conditions examined. 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.

Experiments were also undertaken to determine what effects the size of the property to be proved has on the running time of the algorithm. Preliminary results indicate that an increase of the property size results in a proportional increase in running time. This is

to be expected, as the underlying graph matching algorithm has to match more elements to determine if the property holds or not.

## 8.2 Industrial Experimentation

Aside from the experiments with the police station transformation we have reported in the previous section, we have applied our technique in the context of the NECSIS project. The experiment regards a DSLTrans transformation that maps between subsets of a proprietary GM metamodel, describing legacy automotive configuration data, and the AUTOSAR metamodel, an open platform shared by car manufacturers. Details of this experiment can be found in [5] and a complete description of the transformation can be found in [22]. This DSLTrans mapping transformation includes 7 rules, distributed among three layers. The set of path conditions for the transformation is built in around 0.8 seconds and includes 3 path conditions. This low number of path conditions is due to the fact that, in order to reach a canonical form for the path condition building algorithm, a pre-processing step (as mentioned in Section 4.2) merges several rules coming from the same layer. In [5] we describe the proof of nine properties (multiplicity invariants, security invariants and pattern contracts) that demonstrate several aspects of the correctness of this mapping transformation. All properties were proved in around 0.02 seconds by our approach, and were expressed using a propositional logic extension to the property language that we present in this paper.

## 8.3 Discussion

The experimental results of verifying the test Police Station transformation portrayed in the graphs of Figure 21 show that, as predicted by complexity Equation (1), both the path condition construction time and the number of created path conditions grow exponentially with the number of rules. In Figure 22 we can also see that property proving time for properties that hold increases linearly with the number of path conditions, as was also theoretically predicted in Section 7.2. Despite the exponential time and space complexities of the path condition construction algorithms, our experiments suggest that real-world sized model transformations can be tractable by employing the optimizations described in Section 7.3. We also believe further optimization opportunities of our algorithms exist and that the number of rules handled by our approach can be driven higher.

The industrial case study presented in Section 8.2 suggests that validation of practical model transformations is not always very computationally expensive. In fact, the

properties we have proved in this industrial case study are of practical use for our partners yet required only seconds to prove.

From the differences in the examples we have presented in this section and from our experience with building DSLTrans transformations we believe that the complexity of verifying real-word model transformations can vary within a wide range. Complexity formula 1 provides us a referential that can be used when evaluating the theoretical and operational complexities of verifying further case study transformations. This complexity is influenced by several parameters that describe the shape of a model transformation, and we believe the study of those parameters in further case studies is very important. Refining Equation (1) will provide better precision in our theoretical estimations and also direct our optimization efforts by understanding what transformation parameters have the highest impact on performance.

## 9 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 verified; and 3) the *verification technique* used.

**Kind of Transformations Considered** DSLTrans is a graph based transformation language and as such shares its principles with languages such as AGG [23], AToM<sup>3</sup> [24], VIATRA2 [25], ATL [26] or VTMS [27]. As mentioned previously, DSLTrans' transformations 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, as these transformations have properties that are easier to understand and analyse. However, termination is undecidable for graph based transformation languages [28]. This problem has led to a number of proposed termination criteria, as well as criteria analysis techniques, for transformations written in graph based transformation languages [29–33]. Confluence is also undecidable for graph based transformation languages [34]. As for termination, several confluence criteria and corresponding analysis techniques have been proposed in the literature [35, 33, 36, 37].

**Verifiable Properties of Transformations** 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 [10]. 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. In 2007, Narayanan and Karsai propose verifying model transformations by structural correspondence [11]. In their approach, structural correspondences are defined as pre-condition/post-condition axioms. As 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 pre-condition/post-condition 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*. In our proposal, we aim at proving structural correspondences for all executions of a model transformation, and base the construction of the properties (or pre-condition/post-condition axioms, using the vocabulary in [11]) 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 [12] 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 an input model 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 [16]. 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 [27]. 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 however that since their approach is based on the generic SWI-Prolog inference engine, there could be a performance bottleneck or the possibility of non-terminating computations. They foresee that a specialised reasoning system might be necessary for their approach to scale.

More recently in 2012 and 2013, Guerra *et al.* [38] have proposed techniques for the automated verification of model transformations based on visual contracts. Their work describes a rich and well-studied language for describing syntactic relations between input and output models. These pre- and post- condition graphs then are transformed into OCL expressions, which are fed into a constraint-solver to generate test input models for the transformation. Their framework algorithm can then test a transformation on a number of these input models, and verify them by the OCL expressions. 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. However the verification technique used by Guerra *et al.* differs fundamentally from ours. Our abstraction over the number of elements of the same type present in the model enables our approach to be exhaustive and allows for correctness proofs, while the approach by Guerra *et al.* is aimed at increasing the level of confidence in a transformation through coverage of test cases. A similar white-box generation approach is also seen in recent work by González and Cabot [39].

Also in 2012 Büttner *et al.* have published their work on the verification of ATL transformations [15, 14]. In [15] 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 that should hold, where the property is expressed as an OCL constraint. As the authors mention, Alloy performs bounded verification and as such it does not guarantee that a counterexample is found if it exists. In [14] Büttner *et al.* aim at proving model syntax relation properties of ATL transformations expressed as pre-condition/post-condition 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 transformation under analysis into additional first order logic axioms. Off-the-shelf SMT solvers such as Z3 and Yices are then used to check whether the pre-condition/post-condition OCL constraints hold.

The approach in [14] 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. However, there are several differences with our approach: the authors’ proofs may require human assistance, depending on the used SAT solver; despite the fact that Büttner *et 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 *et al.*’s axiomatization 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 of results such as counterexamples and could be in general less error-prone due to the absence of an indirection layer which maps transformation concepts to concepts in the chosen logic. It is interesting to notice that similar to our approach Büttner *et al.* have chosen *expressiveness reduction* as a means to work with subset of ATL that is verifiable.

Assertional reasoning in graph transformations has been studied by Habel and colleagues, who have introduced nested conditions as properties of graphs in [40]. The authors formally prove these nested conditions have the expressiveness of first-order graph formulas. Poskitt and Plump later propose in [41] a Hoare-style verification calculus which is anchored on their experimental graph programming language GP. Using this calculus they then go on to prove nested condition properties of a graph-colouring GP program. Our approach shares some resemblances with assertional reasoning in that we also propose a pre-condition/post-condition language and a calculus for proving such properties in DSLTrans. We remark however that the theoretical work in assertional reasoning described above is larger in scope than what we present here and

that assertional reasoning results require lifting to more usable graph transformation languages than GP before they can be used in practice.

**Verification Technique Used** A different possibility for our work would have been to utilise the GROOVE tool [42]. 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 behaviour is simulated by graph transformation rules that manipulate those graphs.

In [43] 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 optimisations we have made in our implementation of our approach by maintaining caches throughout path condition 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 could make 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 infeasible to implement our approach by relying solely on GROOVE’s graphical interface.

Still in the context of GROOVE, several studies [44–46] have been performed on abstractions that allow coping with the state space explosion when performing model checking of state-based systems modeled as graph transformations. The authors present various abstractions on state graphs that allow reducing their size during model checking while allowing equivalent (or approximate versions of) proofs of temporal logic properties using the abstracted state graphs. Although our technique is also based on abstraction, our main purpose is not to execute concrete graphs in order to examine the state they represent. We rather implicitly execute all graphs which are in an abstraction relation with the path conditions such that we are able to examine the relations between all of the transformation’s inputs and outputs.

Also from the *verification technique* viewpoint, Becker *et al.* propose a technique for checking a dynamic system where state is encoded as a graph [47]. 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. The applicability or efficiency of Becker *et al.*'s technique when applied to the verification of model syntax relations in model transformations remains to be studied.

## 10 Conclusion

In this paper we have modified symbolic execution techniques to verify DSLTrans model transformations. We have presented an algorithm to prove model syntax relation properties by building all possible path conditions for a transformation.

The concrete contributions of our work are the following:

- An algorithm for constructing all path conditions representing all executions of a DSLTrans transformation
- A property-checking algorithm that proves model syntax relation properties over all path conditions, and therefore over all transformation executions
- Validity and completeness proofs for the path condition construction and property proof algorithms
- A discussion of optimisations and scalability concerns for our methods, along with results from an industrial application

A current area of research interest is the expressiveness of the property language presented in this paper. We have recently completed a propositional logic extension, which has already been used to express and prove meaningful properties in our industrial case study [5]. Another extension to the property language would be the addition of element attributes. This would allow the property designer to specify attribute constraints in the property.

A further topic of interest is that of negative DSLTrans constructs, where elements and associations of given types are prevented from being matched by a rule.

We are also exploring the issue of multiplicity in our verification technique, such as determining for a path condition how many times a particular rule has executed. This information may arise from recording how many times the rule was “glued” to the path condition in the combination step. This could enable properties to be proved over the number of times a rule has executed.

As seen in the results section, we believe our algorithms to scale to industrial-sized case studies. In future work, we will apply our technique to transformations and case studies such as structural relations in Simulink [48], as well as those of interest to our partners in the automotive domain.

## Acknowledgements

This work has been developed in the context of the NECSIS project, funded by Automotive Partnership Canada.

## References

1. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software* **20** (September 2003) 42–45
2. Mens, T., Van Gorp, P.: A Taxonomy of Model Transformations. *Electronic Notes in Theoretical Computer Science* **152** (March 2006) 125–142
3. Amrani, M., Lúcio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Le Traon, Y., Cordy, J.: A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In: *ICST, IEEE* (2012) 921–928
4. Lúcio, L., Barroca, B., Amaral, V.: A Technique for Automatic Validation of Model Transformations. In: *MoDELS, Springer* (2010) 136–150
5. Selim, G., Lúcio, L., Cordy, J.R., Dingel, J.: Symbolic Model Transformation Property Prover for DSLTrans. Technical Report 2013-616, Queen’s University (2013) <http://research.cs.queensu.ca/TechReports/Reports/2013-616.pdf>.
6. Barroca, B., Lúcio, L., Amaral, V., Félix, R., Sousa, V.: DSLTrans: A Turing Incomplete Transformation Language. In: *SLE, Springer* (2010) 296–305
7. Félix, R., Barroca, B., Amaral, V., Sousa, V. Technical report, UNL-DI-1-2010, UNL, Portugal (2010) <http://solar.di.fct.unl.pt/twiki5/pub/Projects/BATIC3S/ModelTransformationPapers/UML2Java.1.zip>.
8. Gomes, C., Barroca, B., Amaral, A.: DSLTrans User Manual <http://msdl.cs.mcgill.ca/people/levi/files/DSLTransManual.pdf>.
9. Zhang, Q., Sousa, V.: Practical Model Transformation from Secured UML Statechart into Algebraic Petri Net. Technical Report TR-LASSY-11-08, U. Luxembourg (2011) <http://msdl.cs.mcgill.ca/people/levi/files/Statecharts2APN.pdf>.
10. Akehurst, D., Kent, S.: A Relational Approach to Defining Transformations in a Metamodel, *Springer* (2002) 243–258
11. Narayanan, A., Karsai, G.: Verifying Model Transformations by Structural Correspondence. *Electronic Communications of the EASST* **10** (2008)
12. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL Contracts for the Verification of Model Transformations. *ECEASST* **24** (2009)

13. Guerra, E., De Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated Verification of Model Transformations based on Visual Contracts. *Automated Software Engineering* **20**(1) (2013) 5–46
14. Büttner, F., Egea, M., Cabot, J.: On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers. In: *MODELS*, Springer (2012) 432–448
15. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL Transformations Using Transformation Models and Model Finders. In: *ICFEM*, Springer (2012) 198–213
16. Asztalos, M., Lengyel, L., Levendovszky, T.: Towards Automated, Formal Verification of Model Transformations. In: *ICST*, IEEE (2010) 15–24
17. Amrani, M., Dingel, J., Lambers, L., Lúcio, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Towards a Model Transformation Intent Catalog. In: *Proceedings of the First Workshop on Analysis of Model Transformations (AMT)*. (October 2012)
18. King, J.: Symbolic Execution and Program Testing. *Communications of the ACM* **19**(7) (1976) 385–394
19. Syriani, E., Vangheluwe, H.: De-/Re-constructing Model Transformation Languages. *ECE-ASST* **29** (2010)
20. Syriani, E., Vangheluwe, H., LaShomb, B.: T-core: a framework for custom-built model transformation engines. *Software and Systems Modeling* (2013) 1–29
21. Lúcio, L.: SyVOLT: A Prototype Implementation (2013) <http://msdl.cs.mcgill.ca/people/levi/files/SyVOLT.zip>.
22. Selim, G., Wang, S., Cordy, J.R., Dingel, J.: Model transformations for migrating legacy models: An industrial case study. In: *Proceedings of ECMFA 2012. Lecture Notes in Computer Science*, Springer (2012) 90–101
23. Taentzer, G.: AGG: A Tool Environment for Algebraic Graph Transformation. In: *AGTIVE. Volume 1779.*, Springer (2000) 333–341
24. De Lara, J., Vangheluwe, H.: ATOM<sup>3</sup>: A Tool for Multi-formalism and Meta-Modelling. In: *FASE '02*, Springer-Verlag (2002) 174–188
25. Varró, D., Pataricza, A.: Generic and Meta-transformations for Model Transformation Engineering. In: *UML*, Springer (2004) 290–304
26. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. *Science of Computer Programming* **72**(12) (2008) 31 – 39
27. Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A Systematic Approach to Meta-modeling Environments and Model Transformation Systems in VMTS. *Electronic Notes in Theoretical Computer Science* **127**(1) (2005) 65–75
28. Plump, D.: Termination of Graph Rewriting is Undecidable. *Fundamentae Informatica* **33**(2) (1998) 201–209
29. De Lara, J., Vangheluwe, H.: Automating the Transformation-based Analysis of Visual Languages. *Formal Aspects of Computing* **22**(3-4) (May 2010) 297–326
30. Ehrig, H.K., Taentzer, G., De Lara, J., Varró, D., Varró-Gyapai, S.: Termination Criteria for Model Transformation. In: *Transformation Techniques in Software Engineering, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany* (2005)
31. Varró, D., Varró-Gyapai, S., Ehrig, H., Prange, U., Taentzer, G.: Termination Analysis of Model Transformations by Petri Nets. In: *ICGT. Volume 4178.*, Springer (2006) 260–274
32. Bruggink, H.J.S.: Towards a Systematic Method for Proving Termination of Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science* **213**(1) (2008)
33. Küster, J.M.: Definition and Validation of Model Transformations. *SoSyM* **5**(3) (2006) 233–259
34. Plump, D.: Confluence of Graph Transformation Revisited. In: *Processes, Terms and Cycles: Steps on the Road to Infinity*, Springer (2005)

35. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: ICGT, Springer (2002)
36. Lambers, L., Ehrig, H., Orejas, F.: Efficient Detection of Conflicts in Graph-based Model Transformation. *Electronic Notes in Computer Science* **152** (2006)
37. Biermann, E.: Local Confluence Analysis of Consistent EMF Transformations. *ECEASST* **38** (2011) 68–84
38. Guerra, E., Soeken, M.: Specification-driven Model Transformation Testing. *Software & Systems Modeling* (2013) 1–22
39. González, C.A., Cabot, J.: Atltest: a white-box test generation approach for atl transformations. In: *Model Driven Engineering Languages and Systems*. Springer (2012) 449–464
40. Habel, A., Pennemann, K.h.: Correctness of high-level transformation systems relative to nested conditions. *Mathematical. Structures in Comp. Sci.* **19**(2) (April 2009) 245–296
41. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. *Fundam. Inform.* **118**(1-2) (2012) 135–175
42. Ghamarian, A., Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and Analysis using GROOVE. *International Journal on Software Tools for Technology Transfer* **14**(1) (2012) 15–40
43. Rensink, A., Schmidt, A., Varró, D.: Model Checking Graph Transformations: A Comparison of Two Approaches. In: ICGT, Springer (2004) 226–241
44. Rensink, A., Distefano, D.: Abstract graph transformation. *Electr. Notes Theor. Comput. Sci.* **157**(1) (2006) 39–59
45. Bauer, J., Boneva, I., Kurbán, M.E., Rensink, A.: A modal-logic based graph abstraction. In: ICGT. Volume 5214 of *Lecture Notes in Computer Science.*, Springer (2008) 321–335
46. Rensink, A., Zambon, E.: Pattern-based graph abstraction. Volume 7562 of *Lecture Notes in Computer Science.*, Springer (2012) 66–80
47. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: ICSE, ACM (2006) 72–81
48. Fehér, P., Mészáros, T., Mosterman, P.J., Lengyel, L.: Flattening Virtual Simulink Subsystems with Graph Transformation

## A Formal Syntax and Semantics of DSLTrans

In this section we will formally introduce the syntax and the semantics of the DSLTrans language. The theory is based on the notion of typed graphs as described in Section 3.

We will start by introducing the notion of *metamodel*, which in DSLTrans is used to type the input and output models of a DSLTrans transformation.

### Definition 28. Metamodel

*A metamodel is a 5-tuple  $\langle V, E, st, \tau, \leq \rangle$  where  $\langle V, E, st, \tau \rangle \in \text{TG}$  is a typed graph,  $(V, \leq)$  is a partial order and  $\tau$  is a bijective typing function. Additionally we also have that: if  $v \in V$  then  $\tau(v) \in VTypes \times \{\text{abstract}, \text{concrete}\}$  where  $VTypes$  is the set of vertex type names; if  $e \in E$  then  $\tau(e) \in ETypes \times \{\text{containment}, \text{reference}\}$ , where  $ETypes$  is a set of edge type names. The set of all metamodels is called META.*

A formal metamodel is particular kind of typed graph where vertices represent classes and edges relations between those classes. A typed graph representing a metamodel has two special characteristics: on the one hand, the typing function for vertices

and edges is bijective. This means that each type occurs only once in the metamodel, as is to be expected. On the other hand a metamodel is equipped with a partial order between vertices. This partial order is used to model specialization at the level of the metamodel's classes. Note that here we have overridden the co-domain of the typing function in the original typed graph presented in Definition 1 in order to allow distinguishing between *abstract* and *concrete* classes, as well as between *containment* and *reference* edges in our metamodels.

**Levi** ► **Cardinalities are not treated, meaning all relations can be n-to-m. It may be necessary to treat cardinalities.** ◀

**Definition 29. Expanded Metamodel**

Let  $mm = \langle V, E, st, \tau, \leq \rangle \in \text{META}$  be a metamodel. The expansion of  $mm$ , noted  $mm^*$ , is a typed graph  $\langle V', E', st', \tau' \rangle \in \text{TG}$  built as follows:

- $V' = V \setminus \{v \in V \mid \tau(v) = (\cdot, \text{abstract})\}$ ;
- $v_1 \xrightarrow{e} v_2 \in E'$  if  $v_1 \xrightarrow{e} v_2 \in E$  and  $\tau(v_1) = (\cdot, \text{concrete})$  and  $\tau(v_2) = (\cdot, \text{concrete})$ ;
- if  $v_1 \xrightarrow{e} v_2 \in E$  we have that:  $v'_1 \xrightarrow{e'} v'_2 \in E'$ , where  $v'_1 \leq v_1$ ,  $v'_2 \leq v_2$  and  $\tau'(e') = \tau(e)$ ;
- for all  $v \in V'$  and  $e \in E'$  we have that  $\tau'(v) = \tau(v)$  and that  $\tau'(e) = \tau(e)$ .

An expanded metamodel is an auxiliary construction where all the relations between types are made explicit, rather than remaining implicit in the specialization hierarchy. It is built by adding to the original metamodel typed graph a relation *rel* between two classes of the metamodel whenever those classes specialize two classes that are related by *rel*. Abstract classes and their relations do not carry over to the expanded metamodel. Expanded metamodels will be used in the subsequent text to facilitate the manipulation of models, transformation rules and other structures as typed graphs.

**Definition 30. Metamodel Instance**

An instance of a metamodel  $mm = \langle V', E', st', \tau', \leq \rangle \in \text{META}$  is a typed graph  $\langle V, E, st, \tau \rangle \in \text{TG}$ , where the codomain of  $\tau$  equals the codomain of  $\tau'$ . Also, there is a typed graph homomorphism  $f : V \rightarrow V'$  from  $\langle V, E, st, \tau \rangle$  to metamodel  $mm^*$  such that the graph  $\langle V, \{v \rightarrow v' \in E \mid \tau(v \rightarrow v') = (\cdot, \text{containment})\}, st \rangle$  is acyclic. The set of all instances for a metamodel  $mm$  is called  $\text{INSTANCE}^{mm}$ .

A metamodel instance is a useful intermediate formal notion that lies between metamodel and model. The injective typed graph homomorphism between a metamodel instance and metamodel models multiple “instances” of objects and relations being typed by one single class or relation of the metamodel. Metamodel instances do not allow the transitive closure of containment relations, as enforced by EMF.

**Definition 31. Containment Transitive Closure**

The containment closure of a metamodel instance  $\langle V', E', st', \tau' \rangle \in \text{INSTANCE}^{mm}$  is a typed graph  $\langle V, E, st, \tau \rangle$  where we have that  $V = V'$ ,  $\tau \subseteq \tau'$  and  $\tau$ 's codomain is the



union of the codomain of  $\tau'$  and the set  $\{indirect\}$ . We also have that  $E' = E \cup E_c^*$  where  $E_c^*$  is the transitive closure of the set  $\{v \xrightarrow{e} v' \mid \tau(v \xrightarrow{e} v') = (\cdot, containment)\}$  and if  $e \in E \setminus E'$  then  $\tau(e) = indirect$ . We note  $mi^*$  the containment closure of a metamodel instance  $mi \in \text{INSTANCE}^{mm}$ .

Given a metamodel instance, its containment transitive closure includes, besides the original graph, all the edges belonging to the transitive closure of containment links in that metamodel instance. The transitive edges are typed as *indirect*.

In definitions that follow we will use the  $*$  notation, as in Definition 31, to denote the containment transitive closure of structures that directly or indirectly include metamodel instances. For example,  $tg^*$  would represent the containment transitive closure of typed graph  $tg$  wherever containment edges are found in the graph.

**Definition 32. Model**

A model of a metamodel  $mm = \langle V', E', st', \tau', \leq \rangle \in \text{META}$  is a metamodel instance  $\langle V, E, st, \tau \rangle \in \text{INSTANCE}^{mm}$ , such that: there exists an injective typed graph homomorphism  $f : V \rightarrow V'$  from  $\langle V, E, st, \tau \rangle$  to metamodel  $mm^*$  where, if there exists an edge  $f(a) \xrightarrow{e'} b \in E'$  where  $\tau(e') = (\cdot, containment)$ , then we also have that  $f(b) \xrightarrow{e} c \in E$  and that  $f(c) = b$ . The set of all models for a metamodel  $mm$  is called  $\text{MODEL}^{mm}$ .

A model, as per Definition 32, is a metamodel instance where all the containment relations are respected. This means that if an object having a containment relation exists in the model, then the model will also contain an instance of that containment relation together with a contained object.

**Levi** ► **this only works if 0 multiplicities are allowed for the metamodel example in the paper.** ◀

**Definition 33. Input-Output Model**

An input-output model is a 6-tuple  $\langle V, E, (s, t), \tau, Input, Output \rangle$ , where:  $Input = \langle V', E', st', \tau' \rangle \in \text{MODEL}^{sr}$  is a model and  $Output = \langle V'', E'', st'', \tau'' \rangle \in \text{INSTANCE}^{tg}$  is a metamodel instance;  $V = V' \cup V''$ ,  $E \subseteq E' \cup E''$  and  $\tau \subseteq \tau' \cup \tau''$ , where the codomain of  $\tau$  is the union of the codomains of  $\tau'$  and  $\tau''$  and the set  $\{trace\}$ . An edge  $e \in E \setminus E' \cup E''$  is called a traceability link and is such that  $s(e) \in V''$ ,  $t(e) \in V'$  and  $\tau(e) = trace$ . The set of all match-apply patterns for a source metamodel  $sr$  and a target metamodel  $tg$  is called  $\text{IOM}_{tg}^{sr}$ .

An input-output model is an object we will use when defining the semantics of a DSLTrans model transformation. As the name suggests, it is composed of two models, one called the *input* and the other one the *output*. An input-output model allows representing intermediate states during the execution of a model transformation. It may include a particular type of edges called *traceability links*, for keeping a history of which elements in the output model originated from which elements in the input model. Note

that while the input part of a input-output model is a model (as per Definition 32), its apply part is a metamodel instance (as per Definition 30). This is so because during a particular moment of in a DSLTans transformation execution the output model may not be well-formed, in the sense that its containment relations are yet not fully respected.

**Definition 34. Metamodel Pattern and Indirect Metamodel Pattern**

A pattern of a metamodel  $mm \in \text{META}$  is an instance of  $mm$ . Given a metamodel pattern  $\langle V', E', st', \tau' \rangle \in \text{INSTANCE}^{mm}$  we have that  $\langle V, E, st, \tau \rangle$  is an indirect pattern if  $V = V'$ ,  $E \subseteq E'$ , the codomain of  $\tau$  is the union of the codomains of  $\tau'$  and the set  $\{\text{indirect}\}$ . Also, if  $v_1 \xrightarrow{e} v_2 \in E \setminus E'$ , then we have that  $\tau(e) = \text{indirect}$ . Given a metamodel  $mm$ , the set of all metamodel patterns for  $mm$  is called  $\text{PATTERN}^{mm}$ . The set of all indirect metamodel patterns for  $mm$  is called  $\text{IPATTERN}^{mm}$ .

Metamodel patterns are introduced in Definition 34 as an auxiliary notion, distinct from the notion of metamodel instance (although formally they are the same), with the goal of introducing new structures that are used to represent rules (and later on path conditions and properties). An *indirect* metamodel pattern is a metamodel pattern that includes edges typed as *indirect*, used in the coming mathematical development to model DSLTrans rules.

**Definition 35. Transformation Rule**

A transformation rule is a 6-tuple  $\langle V, E, (s, t), \tau, \text{Match}, \text{Apply} \rangle$ , where:  $\text{Match} = \langle V', E', st', \tau' \rangle \in \text{IPATTERN}^{sr}$  such that  $\text{Match} \neq \varepsilon^3$  is a non-empty indirect metamodel pattern;  $\text{Apply} = \langle V'', E'', st'', \tau'' \rangle \in \text{PATTERN}^{tg}$  such that  $\text{Apply} \neq \varepsilon$  is a metamodel pattern. We also have that  $V = V' \cup V''$ ,  $E \subseteq E' \cup E''$  and  $\tau \subseteq \tau' \cup \tau''$ , where the codomain of  $\tau$  is the union of the codomains of  $\tau'$  and  $\tau''$  and the set  $\{\text{backward}\}$ . An edge  $e \in E \setminus E' \cup E''$  is called a backward link and is such that  $s(e) \in V''$ ,  $t(e) \in V'$  and  $\tau(e) = \text{backward}$ . We additionally impose that there always exists a  $v_1 \in V''$  in the Apply part of the rule such that  $\nexists e : v_1 \xrightarrow{e} v_2$  and  $\tau(e) = \text{backward}$ . The set of all transformation rules for a source metamodel  $sr$  and a target metamodel  $tg$  is called  $\text{RULE}_{tg}^{sr}$ .

A transformation rule includes a non-empty match pattern and a non-empty apply pattern (also known in the model transformation literature as a rule's *left hand side* and *right hand side*). The apply pattern of a rule always contains at least one apply element that is not connected to a backward link, meaning in practice that a rule will always produce something and not only match. A match pattern can include indirect links that are used to transitively match a model. An apply pattern does not include indirect links as it is used only for the construction of parts of instances of a metamodel. A transformation rule includes backward links, as informally introduced in Section 2.2. Backward links are formally typed as *backward*.

<sup>3</sup> We use the simplified  $\varepsilon$  notation to denote empty n-tuples structures.

**Definition 36. Backward Matcher for a Transformation Rule**

Let  $rl = \langle V, E, st, \tau, Match, Apply \rangle$  be a transformation rule where  $Match = \langle V_m, E_m, st_m, \tau_m \rangle$ . We define  $rl$ 's backward matcher version, noted  $rl^{backM}$ , as the transformation rule  $\langle V', E', st', \tau', Match, Apply' \rangle \sqsubseteq rl$  where  $v_1 \xrightarrow{e} v_2 \in E'$  if and only if  $v_1, v_2 \in Match$  or  $\tau(e) = backward$ .

Definition 35 introduces the notion of backward matcher for a transformation rule which consists solely of the match pattern of a rule and its backward links, if any. The backward matcher of a rule constitutes the complete pattern that a DSLTrans rule attempts to match over an input-output model during rule execution. We will see in the next section that traceability links, generated during execution between the input and output model, are matched by transformation rules' backward links.

**Definition 37. Expanded Transformation Rule**

Let  $rl = \langle V, E, st, \tau, Match, Apply \rangle \in \text{RULE}_{tg}^{sr}$  be a transformation rule where  $Match = \langle V', E', st', \tau' \rangle$  and also we have that  $sr = \langle V'', E'', st'', \tau'', \leq \rangle$ . The expansion of  $rl$ , noted  $rl^*$  is a set of transformation rules built as follows:

- $rl \in rl^*$ ;
- $\langle V, E, st, \tau', Match, Apply \rangle \in rl^*$  iff for all  $v \in V'$  we have that  $\tau'(v) \leq \tau(v)$ .

The expansion of a transformation rule is a set of transformation rules. Each rule in that set includes a possible replacement of each of the classes in the match part of the original rule by one of its subtypes. Expanded transformation rules will be important when defining the semantics of DSLTrans such that polymorphism is correctly handled during rule matching.

**Definition 38. Layer, Transformation**

A layer is a finite set of transformation rules  $l \subseteq \text{RULE}_{tg}^{sr}$ . The set of all layers for a source metamodel  $sr$  and a target metamodel  $tg$  is called  $\text{LAYER}_{tg}^{sr}$ . A transformation is a finite list of layers denoted  $[l_1 :: l_2 :: \dots :: l_n]$  where  $l_k \in \text{LAYER}_{tg}^{sr}$  and  $1 \leq k \leq n$ . We also impose that for any two rules  $rl_1, rl_2 \in \bigcup_{1 \leq k \leq n} l_k$  we never have that  $rl_1^{backM} \cong rl$  and  $rl \sqsubseteq rl_2^{backM}$ , or  $rl_2^{backM} \cong rl$  and  $rl \sqsubseteq rl_1^{backM}$ . The set of all transformations for a source metamodel  $s$  and a target metamodel  $t$  is called  $\text{TRANSF}_{tr}^{sr}$ .

Definition 38 formalizes the notion of a DSLTrans transformation, introduced at the beginning of this section. As expected, a formal DSLTrans transformation is composed of a sequence of layers where each layer is composed of a set of rules. The last condition of Definition 38 imposes that, for any two rules in the transformation, the backward matcher of one rule never partially or totally subsumes (or contains) the backward matcher of the other.

**Notation** In order to achieve a uniform and understandable notation throughout the text that follows we will often abbreviate input-output models (Definition 33) and transformation rules (Definition 35) to their typed graph components. We will also use a simplified notation to refer to the components of the input/output or match/apply typed graphs of those structures. For example, we will abbreviate transformation rule  $\langle V, E, st, \tau, Match, Apply \rangle \in \text{RULE}_{tg}^{sr}$  to only its typed graph  $\langle V, E, st, \tau \rangle$ . In this case we will refer to the transformation rule's vertices that belong to the match part of the graph as  $Match(V)$ , its edges as  $Match(E)$  and so on for pair  $st$  and function  $\tau$ . In a similar fashion we use the notation  $Apply(V)$ ,  $Apply(E)$ , etc. to refer to the transformation rule's components that belong to the apply part of the graph. We will also use the natural extensions of the notions of typed graph union ' $\sqcup$ ' (Definition 2), typed graph homomorphism ' $\triangleleft, \blacktriangleleft$ ' (Definition 3), typed subgraph ' $\sqsubseteq$ ' (Definition 4) and typed graph isomorphism ' $\cong$ ' (Definition 5) to input-output models and transformation rules.

**Transformation Language Semantics** We will now address the semantics of the DSLTrans language. We will start by defining a match function that, given an input-output model and a transformation rule, returns all subgraphs of that input-output model where the rule's match pattern (including backward links) is found.

**Definition 39. Match Function**

Let  $m_{in} \in \text{IOM}_{tr}^{sr}$  be a input-output model and  $rl \in \text{RULE}_{tg}^{sr}$  be a transformation rule. The  $match : \text{IOM}_{tg}^{sr} \times \text{RULE}_{tg}^{sr} \rightarrow \mathcal{P}(\text{IOM}_{tg}^{sr})$  function is defined as follows:

$$match_{rl}(m_{in}) = \{ glue_{noind} \mid glue \sqsubseteq m_{in}^* \wedge glue \cong rl_{trace}^{backM} \}$$

where  $glue$  is an input-output model. Additionally we have that:  $rl_{trace}$  is a version of  $rl$  where edges of type backward have been replaced by edges of type trace;  $glue_{noind}$  is a version of  $glue$  where the indirect links have been removed.

The match function in Definition 39 looks for subgraphs ( $glue \sqsubseteq m_{in}^*$ ) of an input-output model that are isomorphic to the backward matcher of the given transformation rule ( $glue \cong rl_{trace}^{backM}$ ). Note that the containment transitive closure of the input-output model ( $m_{in}^*$ ) is considered such that indirect links in the rule can be looked for. Additionally, indirect links need to be removed from the input-output models resulting from the match function ( $glue_{noind}$ ). This is done because indirect links are not part of the original input model but rather an auxiliary structure.

Let us now turn our attention to the apply function in Definition 40. Its role is extend each of the sub-models found by the a match function for a given input-output model

and a given transformation rule, such that they become isomorphic to the complete rule (minus its backward links). This process effectively creates the new objects and relations specified in the apply part of the rule for each of the sub-models found by the match function.

**Definition 40. Apply Function**

Let  $m_{glue} \in \text{IOM}_{tg}^{sr}$  be a input-output model and  $rl \in \text{RULE}_{tg}^{sr}$  a transformation rule. The  $\text{apply} : \text{IOM}_{tg}^{sr} \times \text{RULE}_{tg}^{sr} \rightarrow \text{IOM}_{tg}^{sr}$  function is defined as follows:

$$\text{apply}_{rl}(m_{in}) = \bigsqcup_{m_{glue} \in \text{match}_{rl}(m_{in})} \text{trace}_{a_{\Delta}}(m_{glue} \sqcup a_{\Delta})$$

where  $a_{\Delta} \in \text{IOM}_{tg}^{sr}$  is such that  $m_{glue} \sqcup a_{\Delta} \cong rl_{noind}$ .

We impose that any instance of  $a_{\Delta}$  is always disjoint from the  $m_{in}$  input-output model and also that any two instances of  $a_{\Delta}$  used in the large union are always disjoint. Partial function  $\text{trace} : \text{IOM} \times \text{IOM} \rightarrow \text{IOM}$  is such that  $\text{trace}_{\langle V_{\Delta}, E_{\Delta}, st_{\Delta}, \tau_{\Delta} \rangle}(\langle V, E, st, \tau \rangle) = \langle V, E', st', \tau' \rangle$  where we have that  $E \subseteq E'$ ,  $st \subseteq st'$  (using a light notational abuse for the  $s \subseteq s'$  and  $t \subseteq t'$ ),  $\tau \subseteq \tau'$  and if  $v_1 \xrightarrow{e} v_2 \in E' \setminus E$  then  $v_1 \in \text{Output}(V_{\Delta})$ ,  $v_2 \in \text{Input}(V)$  and  $\tau'(e) = \text{trace}$ . Finally,  $rl_{noind}$  is a version of  $rl$  where indirect links have been removed.

In Definition 40  $a_{\Delta}$  is an input-output model that contains an instance of the target metamodel. These instances are created by rule  $rl$  and are used to extend the sub-models found by the match function. The  $\text{trace}$  function builds traceability edges between vertices of the output model in  $a_{\Delta}$  and all the vertices from the input part of a model fragment found by the match function.

Note that, because we do not pose any constraints on  $a_{\Delta}$  other than the fact that its union with the sub-model  $m$  is isomorphic to  $rl_{noind}$ , the  $a_{\Delta}$  variable can always be satisfied by an unlimited amount of input-output models. In order to avoid an infinite amounts of results when a transformation rule is executed, in what follows we will consider transformation results differ only up to typed graph isomorphism.

Definitions 39 and 40 are complementary: the former gathers all the sub-models of an input-output model that are matched by a transformation rule; the latter glues on the output part of each of those sub-models new objects and relations created by a transformation rule.

**Definition 41. Layer Step Semantics**

Let  $l \in \text{LAYER}_{tg}^{sr}$  be a Layer. The layer step relation  $\xrightarrow{\text{layerstep}} \subseteq \text{IOM}_{tg}^{sr} \times \text{IOM}_{tg}^{sr} \times \text{LAYER}_{tg}^{sr} \times \text{IOM}_{tg}^{sr}$  is defined as follows:

$$\overline{\langle m_{in}, m_{glue}, \epsilon \rangle} \xrightarrow{\text{layerstep}} m_{in} \sqcup m_{glue}$$

$$\frac{rl \in l, \text{apply}_{rl}(m_{in}) = m_{rout}, \langle m_{in}, m_{glue} \sqcup m_{rout}, l \setminus \{rl\} \rangle \xrightarrow{\text{layerstep}} m_{out}}{\langle m_{in}, m_{glue}, l \rangle \xrightarrow{\text{layerstep}} m_{out}}$$

where  $m_{rout} \in \text{IOM}_{tg}^{sr}$  and  $rl \in \text{RULE}_{tg}^{sr}$ .

We impose that all input-output models that are part of  $rout$  and have been generated by rule  $rl$  are disjoint from input-output models accumulated in  $m_{glue}$  that have been generated by other rules.

In Definition 41 we build the result of executing a layer of a DSLTrans transformation. The operational semantics-like rules in the definition execute each rule  $rl$  in layer  $l$ , in a non-deterministic order, by using the *apply* function. The result of executing each rule is accumulated in the temporary  $m_{glue}$  input-output model. Finally, when the set of transformation rules in the layer has been exhausted, the result of executing all the rules in the layer (now contained in  $m_{glue}$ ) is united with the input-output model  $m_{in}$ , the input to layer  $l$ . Note that this final union produces the result we expect because of the fact that the  $m_{glue}$  input-output model is not disjoint from  $m_{in}$ . The common “glue” parts of  $m_{glue}$  that have been built by the match function and extended by the apply function are now used to build the result of executing layer  $l$ .

Definition 41 is the core of DSLTrans’ semantics. Many model transformation languages are based on graph rewriting, where the result of each rule rewrite is immediately usable by all other rules. In DSLTrans the result of executing one layer in DSLTrans is totally produced before the input to the layer is changed. This is enforced in Definition 41 by the fact that the apply function always executes over the same  $m_{in}$  input-output model and all the results of rule execution in the same layer are added to the  $m_{glue}$  structure that is write-only. Rules belonging to the same layer are thus forced to execute independently.

**Definition 42.** *Transformation Step Semantics*

Let  $[l :: tr] \in \text{TRANSF}_{tg}^{sr}$  be a transformation, where  $l \in \text{LAYER}_{tg}^{sr}$  is a Layer and  $tr$  also a transformation. The transformation step relation  $\xrightarrow{\text{trstep}} \subseteq \text{IOM}_{tg}^{sr} \times \text{TRANSF}_{tg}^{sr} \times \text{IOM}_{tg}^{sr}$  is defined as follows:

$$\frac{\langle m, [] \rangle \xrightarrow{\text{trstep}} m}{\langle m_{in}, \epsilon, l^* \rangle \xrightarrow{\text{layerstep}} m_{inter}, \langle m_{inter}, R \rangle \xrightarrow{\text{trstep}} m_{out}} \quad \text{where } l^* = \bigcup_{rl \in l} rl^*$$

$$\frac{\langle m_{in}, [l :: T] \rangle \xrightarrow{\text{trstep}} m_{out}}{\langle m_{in}, \epsilon, l^* \rangle \xrightarrow{\text{layerstep}} m_{inter}, \langle m_{inter}, R \rangle \xrightarrow{\text{trstep}} m_{out}}$$

While the execution of the rules belonging to a layer happens in parallel, the execution of the layers of a transformation happens sequentially. As per Definition 42, the input-output model  $m_{inter}$  is the output of executing a given layer that is passed onto the next layer as input. The transformation rules in a layer are expanded before execution ( $l^*$ ) such that polymorphism in match elements can be handled (see Definition 37).

**Definition 43. Model Transformation Execution**

Let  $input \in \text{MODEL}^{sr}$  and  $output \in \text{MODEL}^{tg}$  be models and  $tr \in \text{TRANSF}_{tg}^{sr}$  be a transformation. Assume we also have that:

$$\langle V, E, st, \tau, input, \epsilon \rangle, tr \xrightarrow{trstep} \langle V', E', st', \tau', input, output \rangle$$

A model transformation execution is the input-output model  $\langle V', E', st', \tau', input, output \rangle$ . The set of all model transformation executions for transformation  $tr$  is written  $\text{EXEC}(tr)$  and the empty model transformation execution is noted  $\epsilon_{ex}$ .

Finally, as stated in Definition 43, we consider a model transformation execution to be the complete input-output model resulting from executing a transformation on an input model. Note that in this result we have the input model, the output model and the traceability links between their elements that are generated during the transformation's execution. Although considering the complete input-output model as the transformation result is interesting for the purposes of this paper, in practice a transformation developer normally only interested in the output component of the input-output model in a transformation execution.

Note that, because in Definition 43 *output* is a formal model, we enforce that containment relations are respected in the transformation's output.

We now prove two important properties about DSLTrans' transformations.

**Proposition 5. Confluence**

Every model transformation is confluent up to typed graph isomorphism.

**Proposition 6. Termination**

Every model transformation terminates.

## B Validity and Completeness of Path Condition Generation

**Proposition 7. (Validity)** Every path condition abstracts at least one transformation execution.

*Proof.* Let  $tr \in \text{TRANSF}_{tg}^{sr}$  be a DSLTrans transformation. We wish to demonstrate that, for all path conditions  $pc \in \text{PATHCOND}(tr)$ , there exists a transformation execution  $ex \in \text{EXEC}(tr)$  of the set of rules used to build  $pc$  such that  $ex \Vdash pc$ . The proof is built by induction: we will show that, if every path condition generated for a transformation

having  $k - 1$  layers of the transformation abstracts at least one transformation execution, then every path condition generated for a transformation having  $k$  layers will also abstract at least one transformation execution.

- *Base case:* the base case is the case when we have  $tr = []$ , the empty transformation. In this case, according to Definition 23 only the empty path condition  $\epsilon_{pc}$  exists in the path condition set. The empty path condition abstracts the empty transformation execution  $\epsilon_{ex}$ , i.e.  $\epsilon_{ex} \Vdash \epsilon_{pc}$ . This is so because two empty typed graph homomorphisms satisfy the conditions of Definition 24 for an injective and a surjective typed graph homomorphism between the empty path condition and the empty transformation execution;
- *Inductive case:* if for all  $pc' \in \text{PATHCOND}([l_1 :: \dots :: l_{k-1}])$  we have that there exists an  $ex' \in \text{EXEC}([l_1 :: \dots :: l_{k-1}])$  such that  $ex' \Vdash pc'$ , then, for all  $pc \in \text{PATHCOND}([l_1 :: \dots :: l_k])$  we have that there exists an  $ex \in \text{EXEC}([l_1 :: \dots :: l_k])$  such that  $ex \Vdash pc$ .

In order to demonstrate the inductive case we need to show the property holds for all path conditions resulting from combining rules from layer  $l_k \in \text{LAYER}_{Ig}^{sr}$  with a path condition of  $pc \in \text{PATHCOND}([l_1 :: \dots :: l_{k-1}])$ . These path conditions are built as expressed in Definition 21 that combines a path condition  $pc \in \text{PATHCOND}([l_1 :: \dots :: l_{k-1}])$  with all rules of layer  $l_k$ . We will again use induction for this proof.

- *Base case:* this is the case where layer  $l_k$  contains no rules. If  $l_k = \emptyset$  we have that  $\text{PATHCOND}([l_1 :: \dots :: l_{k-1} :: l_k]) = \text{PATHCOND}([l_1 :: \dots :: l_{k-1}])$  because, by the base case of definition 21, no new path condition is added to the set of path conditions generated for the transformation  $[l_1 :: \dots :: l_{k-1}]$ . We know by the hypothesis of the previous induction the property holds for all path conditions in  $\text{PATHCOND}([l_1 :: \dots :: l_{k-1}])$ . Consequently the property trivially holds for all path conditions in  $\text{PATHCOND}([l_1 :: \dots :: l_{k-1} :: \emptyset])$ ;
- *Inductive case:* for the the inductive case (transitive case of ??) we need to show that, assuming the property holds for all path condition  $pc \in PC$  such that  $\langle \epsilon_{pc}, [l_1 :: \dots :: l_{k-1} :: l_k] \rangle \xrightarrow{\text{pathcondgen}} PC$ , then the property also holds for all path condition  $pc' \in PC'$  such that  $\langle \epsilon_{pc}, [l_1 :: \dots :: l_{k-1} :: l_k \cup rl] \rangle \xrightarrow{\text{pathcondgen}} PC'$ , where we have added a rule  $rl$  to  $l_k$  (where  $rl \in \text{RULE}_{Ig}^{sr}$ ). We will thus need to consider three cases of rule combination:

Rule  $rl$  has **partially satisfied dependencies**. By Definition 19 of the  $p_{comb}$  (partial combination) relation, a new set of path conditions  $PC' = \bigcup_{pc \in PC} PC \cup pc \stackrel{\text{trace}}{\sqcup} ma_{\Delta}$  is added to the final path condition set, where  $PC = \text{PATHCOND}([l_1 :: \dots :: l_{k-1} :: l_k])$ . Given we know by hypothesis that for all  $pc' \in PC$  the property holds, we thus need to show that every path condition  $pc' \in \bigcup_{pc \in PC} PC \cup pc \stackrel{\text{trace}}{\sqcup} ma_{\Delta}$  also abstracts at least one transformation execution of  $\text{EXEC}([l_1 :: \dots :: l_{k-1} :: l_k \cup rl])$ . Assuming any  $pc' \in PC'$  we need to show, according to Definition 24, two propositions:

1.  $\forall rl \in \text{Rules}_{pc'} . \text{Match}(rl) \triangleleft \text{Input}(ex')^*$ , and
2.  $(ex')^{\text{traceOut}} \blacktriangleleft (pc')^{\text{traceApply}}$ .



Let us start by demonstrating Item 1. Consider an input-output model  $iom = \langle V, E, st, \tau, Input, \varepsilon \rangle \in \text{IOM}_{lg}^{sr}$  where  $Input \in \text{INSTANCE}^{sr}$  is such that  $match_{rl}(Match) \neq \emptyset$ , i.e. rule  $rl$  matches over the input of  $iom$  (see Definition 39). Because DSLTrans' rules in the same layer of a transformation only match over the result of the previous layer (see Definition 41), we also know that  $rl$  can execute and necessarily there exists a transformation execution  $ex' = apply_{rl}(ex \sqcup iom) \in \text{EXEC}([l_1 :: \dots :: l_{k-1} :: l_k \cup rl])$ . By Definition 39 we also know there exists a typed graph isomorphism between the backward matcher  $rl_{trace}^{backM}$  and a subgraph of  $iom^*$ . Because by Definition 36 we know that  $Match(rl) \sqsubseteq rl_{trace}^{backM}$  and because a typed graph isomorphism is an injective typed graph homomorphism, we know that  $Match(rl) \triangleleft iom^*$ . Consequently we can say that  $Match(rl) \triangleleft Input(ex \sqcup iom)^*$ , given that  $iom^* \sqsubseteq ex^* \sqcup iom^* \sqsubseteq (ex \sqcup iom)^*$ . Because by hypothesis we know that for all rules  $rl \in Rules(pc)$  we have that  $Match(rl) \triangleleft Input(ex)^*$  and we know by Definition 24 that  $Rules(pc') = Rules(pc) \cup rl$  we finally have that  $\forall rl \in Rules_{pc'} . Match(rl) \triangleleft Input(ex \sqcup iom)^*$  and we have finished the proof of Item 1.

In order to demonstrate Item 2, that a surjection exists between  $(ex')^{traceOut}$  and  $(pc')^{traceApply}$  we will start from the induction hypothesis, i.e. that there exists a surjection exists between  $ex^{traceOut}$  and  $pc^{traceApply}$ . By Definition 3 we know a typed graph homomorphism between  $g$  and  $g'$  is a function  $f : V \rightarrow V'$  such that for all  $v_1 \xrightarrow{e} v_2 \in E$  we have that  $f(v_1) \xrightarrow{e'} f(v_2) \in E'$  where  $\tau(v_1) = \tau'(f(v_1))$ ,  $\tau(v_2) = \tau'(f(v_2))$  and also  $\tau(e) = \tau(e')$ . By this definition we then know that if we add to  $g$  a graph  $h$  with a single joint or disjoint edge  $v_1 \xrightarrow{e} v_2$  with  $E$  and to  $g'$  a graph  $h'$  a single joint or disjoint edge  $v'_1 \xrightarrow{e'} v'_2$  to  $E'$  such that  $\tau(v_1) = \tau'(v'_1)$ ,  $\tau(v_2) = \tau'(v'_2)$  and  $\tau(e) = \tau(e')$ , then  $f \cup (v_1, v'_1), (v_2, v'_2)$  is a surjective typed graph homomorphism between  $g \sqcup h$  and  $g' \sqcup h'$ . We will use this fact in our proof of Item 2 to argue that, because for every arc we add to  $ex^{traceOut}$  when executing  $rl$  there is similarly typed arc added to  $pc^{traceApply}$  when  $pc$  is combined with  $rl$ , then a typed graph surjective homomorphism between exists  $(ex')^{traceOut}$  and  $(pc')^{traceApply}$  and can be obtained as explained above.

Let us then analyse what edges are added to  $ex^{traceOut}$  when  $rl$  is executed. We a rule  $rl$  executes once over  $ex \sqcup iom$  then the set  $match_{rl}(ex \sqcup iom)$  has only one element which we will call  $m_{glue}$ . We know that  $m_{glue} \cong rl^{backM}$  and by Definition 40 of apply function and by Definition 41 of the semantics of layer execution we know that  $ex' = ex \sqcup trace_{a_\Delta}(m_{glue} \sqcup a_\Delta)$ , where Let us then break down  $trace_{a_\Delta}(m_{glue} \sqcup a_\Delta)$  in its components edges and show that corresponding edges already exist in  $pc$ , or are added in  $pc^{trace} \sqcup ma_\Delta$ . Note that, if rule  $rl$  executes more than once the result below still holds given that, by Definition 40 copies of  $ma_\Delta$  are added  $ex$  over different  $m_{glue}$  links.

- Let us first consider the  $m_{glue}^{traceOut}$  edges. By Definition 39 of match function we know that  $m_{glue} \sqsubseteq ex$  (i.e.  $m_{glue}$  is joint with  $ex$ ). Because by hypothesis we know that  $ex^{traceOut} \triangleleft pc^{traceApply}$  and by Definition 24 that  $m_{glue}^{traceOut} \sqsubseteq ex^{traceOut}$ , the  $m_{glue}^{traceOut}$  edges are already mapped by the typed graph surjective homomorphism between  $ex$  and  $pc$  and nothing else needs to be shown.

- Let us now consider the  $a_{\Delta}^{traceOut}$  edges. From Definition 40 we know that  $a_{\Delta}$  is such that  $m_{glue} \sqcup a_{\Delta} \cong rl_{noind}$ . We also know from Definition 19 that  $pc' = pc \sqcup^{trace} ma_{\Delta}$  and that  $ma_{\Delta} \sqcup rl_{glue}$  is isomorphic to  $rl$ . Because we are only interested in  $a_{\Delta}^{traceOut}$  and the apply part of  $rl_{noind}$  is preserved when the indirect links are removed, because typed graph isomorphism composition results in a typed graph isomorphism, we have that  $a_{\Delta}^{traceOut}$  is isomorphic to  $ma_{\Delta}^{traceOut}$ . We thus know by Definition 3 of typed graph isomorphism that there exists a function  $f$  from the edges of  $a_{\Delta}^{traceOut}$  into the edges of  $ma_{\Delta}$  such that for every edge  $v_1 \xrightarrow{e} v_2$  of  $a_{\Delta}^{traceOut}$  we have an edge  $f(v_1) \xrightarrow{e'} f(v_2)$  of  $ma_{\Delta}$  where the typing of the vertices and the edges is preserved. As was previously shown, we know that, because  $ex^{traceOut} \blacktriangleleft pc^{traceApply}$  then if we have a graph  $g$  containing a single edge  $v_1 \xrightarrow{e} v_2$  of  $a_{\Delta}$  and another graph  $g'$  containing a single edge  $f(v_1) \xrightarrow{e'} f(v_2)$  we have that  $ex^{traceOut} \sqcup g \blacktriangleleft pc^{traceApply} \sqcup g'$ . If we now add the remaining edges of  $a_{\Delta}$  and  $ma_{\Delta}$  to  $ex^{traceOut}$  and  $pc^{traceApply}$  in the same manner we can deduce that  $ex^{traceOut} \sqcup (m_{glue} \sqcup a_{\Delta}) \blacktriangleleft pc^{traceApply} \sqcup ma_{\Delta}$ .
- Finally we need to consider the traceability edges added on the execution by  $trace_{a_{\Delta}}(m_{glue} \sqcup a_{\Delta})$  and on the path condition by the  $\sqcup^{trace}$  operator in  $pc \sqcup^{trace} ma_{\Delta}$ . From the previous point we know that  $ma_{\Delta} \sqcup rl_{glue} \cong rl$  and that  $m_{glue} \sqcup a_{\Delta} \cong rl_{noind}$ , thus that  $ma_{\Delta} \sqcup rl_{glue} \cong (m_{glue} \sqcup a_{\Delta})_{noind}$ . From Definition 40 we know that  $trace_{a_{\Delta}}(m_{glue} \sqcup a_{\Delta})$  adds edges  $v_1 \xrightarrow{e} v_2$  with type  $trace$  to  $m_{glue} \sqcup a_{\Delta}$  if  $v_1$  is a vertex of  $a_{\Delta}$  and  $v_2$  is a vertex of  $Input(m_{glue} \sqcup a_{\Delta})$ . On the other hand, by Definition 15 we know that  $pc \sqcup^{trace} ma_{\Delta}$  unites to  $pc$  the  $ma_{\Delta}$  rule and that the  $v_1 \xrightarrow{e} v_2$  edges with type  $trace$  are added to  $pc \sqcup^{trace} ma_{\Delta}$  if  $v_1$  is a vertex of  $Apply(ma_{\Delta})$  but not a vertex of  $Apply(pc)$  and  $v_2$  is a vertex of  $ma_{\Delta}$ . Because  $ma_{\Delta} \sqcup rl_{glue} \cong (m_{glue} \sqcup a_{\Delta})_{noind}$  we know that for every  $v_1 \xrightarrow{e} v_2$  with type  $trace$  added by  $trace_{a_{\Delta}}(m_{glue} \sqcup a_{\Delta})$  there is a corresponding traceability arc in  $pc \sqcup^{trace} ma_{\Delta}$ . By definition of surjective typed graph homomorphism we can then deduce that  $(ex \sqcup trace_{a_{\Delta}}(m_{glue} \sqcup a_{\Delta}))^{traceOut} \blacktriangleleft (pc \sqcup^{trace} ma_{\Delta})^{traceApply}$ , and we have concluded the proof of Item 2.

The cases where the rule  $rl$  has **totally satisfied dependencies** (Definition 19) and **no dependencies** (Definition 17) are covered by our proof above. The case where rule  $rl$  has **unsatisfied dependencies** (Definition 18) is trivial given no new path conditions are generated.

**Proposition 8.** (Completeness) *Every transformation execution is abstracted by one path condition.*

*Proof.* blah

**Lemma 2.** *A Transformation Execution is Abstracted by Exactly One Path Condition*

*Proof.* blah

## C Validity and Completeness of Property Verification

**Proposition 9.** (Validity) *The result of checking a property on a path condition and of checking the property on all the transformation executions that path condition abstracts will be the same.*

Let  $t$  be a DSLTrans transformation having source metamodel  $s$  and target metamodel  $t$ . Let also  $ex = \langle V_x, E_x, \tau_x, Match_x, Apply_x, Tl_x \rangle \in Exec_t^s$  be a transformation execution of a transformation  $t$ ,  $pc = \langle V_{pc}, E_{pc}, \tau_{pc}, Match_{pc}, Apply_{pc}, Tl_{pc}, Il_{pc} \rangle \in PC_t^s$  be a path condition of transformation  $t$  and  $p = \langle V_p, E_p, \tau_p, Match_p, Apply_p, Bl_p, Il_p \rangle \in Property_t^s$  be a property of  $t$ . This given, we have that:

$$ex \Vdash pc \wedge pc \vdash p \iff ex \models p$$

**Levi** ► *If a property holds on the path condition then for all transformation executions abstracted by the path condition, the property also holds.* ◀

*Proof.* The proof of our proposition is divided in two lemmas: Lemma 3, the  $ex \Vdash pc \wedge pc \vdash p \implies ex \models p$  direction of the equivalence; and Lemma 4, the  $ex \models p \implies ex \Vdash pc \wedge pc \vdash p$  direction of the equivalence.  $\square$

**Lemma 3.** *If a property holds for a path condition then the property holds for any transformation execution that path condition abstracts.*

*Proof.* In order to prove that  $ex \Vdash pc \wedge pc \vdash p \implies ex \models p$  we have to demonstrate that, if there exists a typed graph injective homomorphism between  $Match_p$  and  $Match_x^*$  then there exists an typed graph injective homomorphism between  $p$  and  $ex^*$ . Throughout the proof we will make use of an example inspired by the police station transformation. The example is given in Figure 23 and presents the three possible artifacts involved in the premises of the proposition: a property  $p$ , a path condition  $pc$  and a transformation execution  $ex$ . The artifacts respect the relations in the premise of the proposition:  $ex \Vdash pc$  and  $pc \vdash p$ . The example will allow us to graphically portray how the several relations needed to demonstrate that  $ex \models p$  are built. As previously introduced in the paper, the  $p$ ,  $pc$  and  $ex$  in Figure 23 are depicted as graphs where the match (or pre-condition, for properties) part are represented on a white background and the apply (or post-condition, for properties) are represented on a grey background. Full arrows represent direct links, while dashed arrows represent indirect links.

This proof will be performed in two parts: in part 1) we will start by showing that the premises of our proposition,  $ex \Vdash pc$  and  $pc \vdash p$ , imply that there exists a typed graph injective homomorphism between  $Match_p$  and  $Match_x^*$ ; in part 2) we then need to show that the premises of our proposition and the fact that there exists a typed graph injective homomorphism between  $Match_p$  and  $Match_x^*$  imply that there exists an typed graph injective homomorphism between  $p$  and  $ex^*$ .

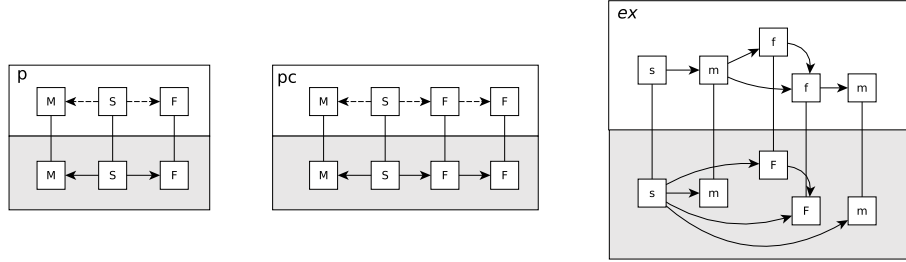


Fig. 23: A Property, a Path Condition and a Transformation Execution for the Police Station Transformation

Regarding part 1), let us show that  $ex \Vdash pc$  and  $pc \vdash p$  imply that there exists a typed graph injective homomorphism between  $Match_p$  and  $Match_x^*$ . Given that  $pc \vdash p$ , by Definition 27 we only have to consider the case where there exists a typed graph injective homomorphism  $f$  between  $Match_p$  and  $Match_{pc}^*$ . Given  $ex \Vdash pc$ , by ?? we only have to consider the case where a typed graph injective homomorphism  $g$  exists between  $Match_{pc}$  and  $Match_x^*$ . Given the above, we can conclude that there exists a typed graph injective homomorphism between  $Match_p$  and  $Match_x^*$  that can be built as the composition of  $g$  and  $f$ , i.e.  $g \circ f$ . This part of the proof is exemplified in Figure 24. Note that in the figure we have represented the transitive closure of  $pc$  and  $ex$  by including a different type of dashed arrows between elements that are transitively linked.

Regarding part 2), let us now show that the existence of a typed graph injective homomorphism between  $Match_p$  and  $Match_x^*$  implies that there exists a typed graph injective homomorphism between  $p$  and  $ex^*$ . We have that  $pc \vdash p$  and as such by Definition 27 there exists a typed graph injective homomorphism  $k$  between  $p$  and  $pc^*$ . We also have that  $ex \Vdash pc$ , which means that  $\langle V_x, E_x \setminus E_{mx}, \tau_x \rangle$  is a typed graph strict instance of  $\langle V_{pc}, E_{pc} \setminus E_{mpc}, \tau_{pc} \rangle$ , where  $Match_x = \langle V_{mx}, E_{mx}, \tau_{mx} \rangle$  and  $Match_{pc} = \langle V_{mpc}, E_{mpc}, \tau_{mpc} \rangle$ .

By ?? this means there exists a surjective graph homomorphism  $h$  between  $\langle V_x, E_x \setminus E_{mx}, \tau_x \rangle$  and  $\langle V_{pc}, E_{pc} \setminus E_{mpc}, \tau_{pc} \rangle$ . Relation  $h^{-1}$ , the inverse of  $h$ , necessarily contains an injective typed graph homomorphism  $i \subseteq h^{-1}$ , given that  $h$  is type respecting. The relations involved in this part of the proof are exemplified in Figure 25. Note that, since  $i$  overlaps with part of  $h^{-1}$ , in order to make it distinct in Figure 25,  $i$  is represented using a thicker dashed line than the other homomorphisms in the picture.

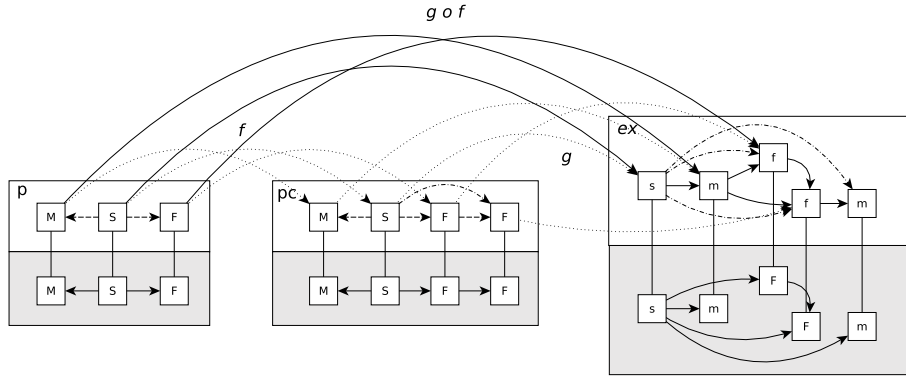


Fig. 24: Building the Injective Homomorphism between the Match part of a Property and a Transformation Execution

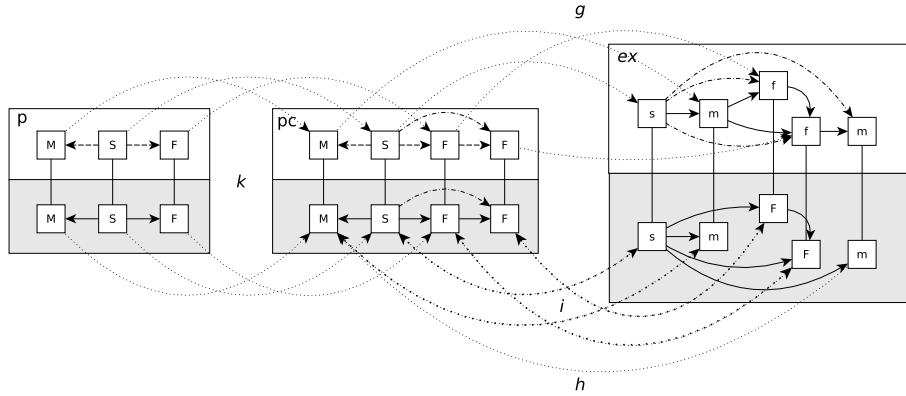


Fig. 25: Building the Injective Homomorphism between the Apply part of a Path Condition and a Transformation Execution

We can thus finally build an injective homomorphism  $j = g \cup i$  between  $pc^*$  and  $ex^*$ . This morphism is exemplified in figure 26.

From the above we can conclude that there exists a typed graph injective homomorphism between  $p$  and  $ex^*$  that can be built as  $j \circ k$ , the composition of injective typed graph isomorphisms  $k$  and  $j$ . This homomorphism is exemplified in Figure 27.

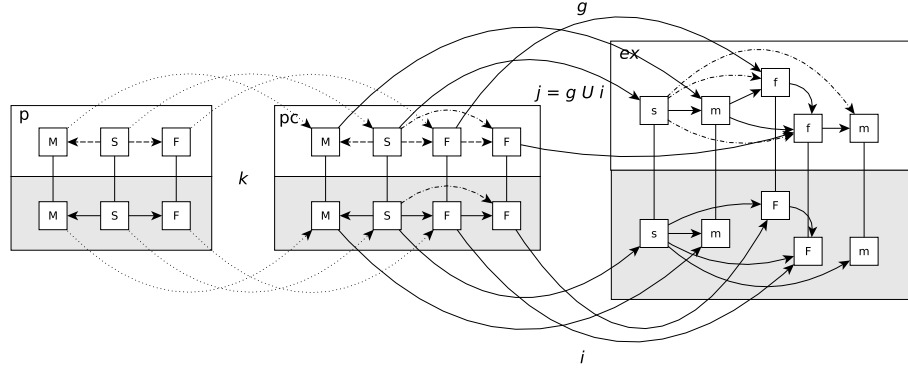


Fig. 26: Building the Injective Homomorphism between a Path Condition and a Transformation Execution

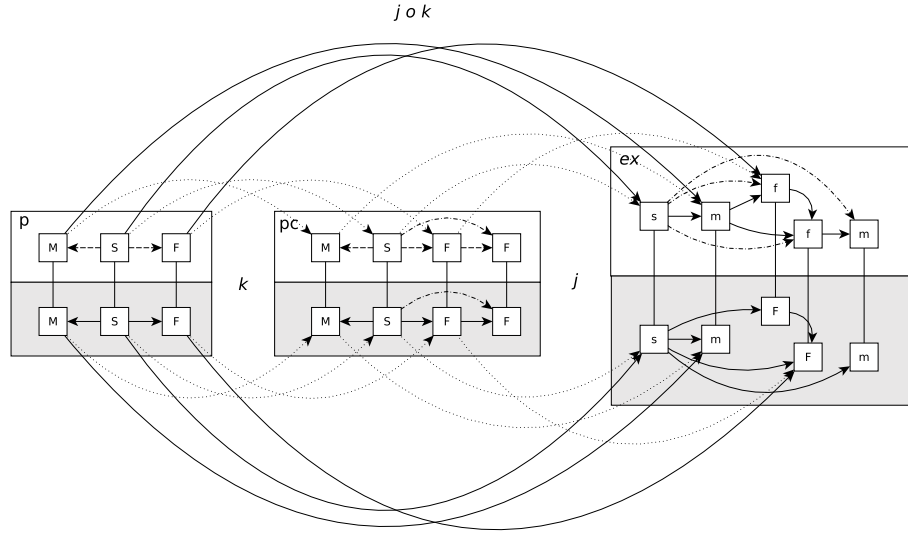


Fig. 27: Building the Injective Homomorphism between a Property and a Transformation Execution

□

**Lemma 4.** *If a property holds for a transformation execution then the property holds for the path condition that abstracts it.*

*Proof.* We will now prove the reverse direction of the implication, i.e. that:

$$ex \models p \implies ex \Vdash pc \wedge pc \vdash p$$

By using contraposition we can show that:

$$\neg(ex \Vdash pc \wedge pc \vdash p) \implies \neg(ex \models p)$$

We only need to worry about the case where we have  $ex \Vdash pc \wedge \neg(pc \vdash p)$  given our property proof results are meaningful only for transformation executions which are abstracted by the path condition being examined. In this case we have that, because  $ex \Vdash pc$ , by ?? we know that there is a surjective typed graph homomorphism between the apply pattern of transformation execution  $ex$  and the apply pattern of the path condition  $pc$ . Given we know that  $\neg(pc \vdash p)$ , then we know that an injective type graph homomorphism does not exist between  $p$  and  $pc$ . This means that, by the Definition 3 of injective typed graph homomorphism, there exists a type  $T$  which is instantiated in  $pc$  but not in  $prop$ . However, because of the fact that  $ex \Vdash pc$  we know that  $T$  is instantiated in  $ex$  because a surjective typed graph homomorphism exists between  $ex$  and  $pc$ , implying type  $T$  is instantiated at least once in  $ex$ . We thus can deduce that we have  $\neg(ex \models p)$  given that  $ex \models p$  implies that there exists an injective type graph homomorphism between property  $p$  and execution  $ex$ . This injective type graph homomorphism cannot exist given the fact an instance of  $T$  exists in  $ex$  but not in  $p$ .  $\square$

**Proposition 10.** *(Completeness) Proving that a property holds or does not hold by examining all path conditions of the transformation is equivalent to proving the property holds or does not hold for all transformation executions.*

*Proof.* blah  $\square$