SoSyM manuscript No.
(will be inserted by the editor)

A Technique for Symbolically Verifying Properties of Graph-Based Model Transformations

Levi LÚCIO[1], Bentley James OAKES[1], and Hans VANGHELUWE[21]

[1] School of Computer Science, McGill University, Canada
[2] University of Antwerp, Belgium

The date of receipt and acceptance will be inserted by the editor

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 has 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. We demonstrate that this property proving approach is both valid and complete. Implementation results are also presented here which suggest that our approach is feasible and can scale to real-world transformations.

Key words 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.* [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 constructs a set of path conditions, where each path condition symbolically 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 [5].

Our approach is feasible due to the use of the transformation language DSLTrans [6]. 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 [4]. Our approach currently considers a core subset of the DSLTrans language that does not include negative conditions in rules or attribute ma-

nipulation. These features of the language will be addressed in future work.

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 correctly transformed into elements of the output model, and have been explored by several authors [10–13]. In our proof technique, the properties examined can be proven to hold for all executions of a given model transformation, no matter the input model. Therefore, our technique is *transformation dependent* and *input independent* [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 [14–16]) but instead work on DSLTrans transformations themselves. Along with DSLTrans rules, all of the constructs involved in our algorithms are typed graphs. This intuitive representation allows our property proving technique to be composed of relatively simple steps, as the metamodels, models, and properties involved are all constructed using a similar graphical representation.

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.

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;
- Validity and completeness proofs of the path condition construction and property proving algorithms;
- A discussion of performance and scalability results for our implementation.

This paper is organised as follows: Section 2 briefly introduces the DSLTrans model transformation language and its formal semantics, while the formal background for this work is presented in Section 3. The algorithms to build the complete set of path conditions for the transformation will be discussed in Section 4. Section 5 will present the abstraction relation found in our technique, along with examples, while Section 6 will examine how this abstraction relation is used in our process for proving properties. In Section 7 and Section 8, 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.

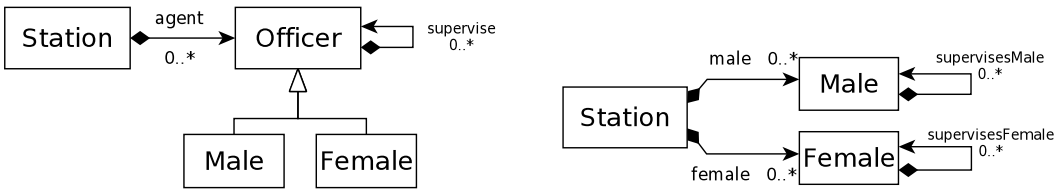
2 The DSLTrans Transformation Language

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 Appendix B.

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

In Figure 2 we present a DSLTrans transformation that involves both metamodels. 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 a transformation execution always terminate, but yet still allows for appropriate expressiveness.



(a) Organization language (b) Gender language

Fig. 1: Metamodels for the Police Station transformation

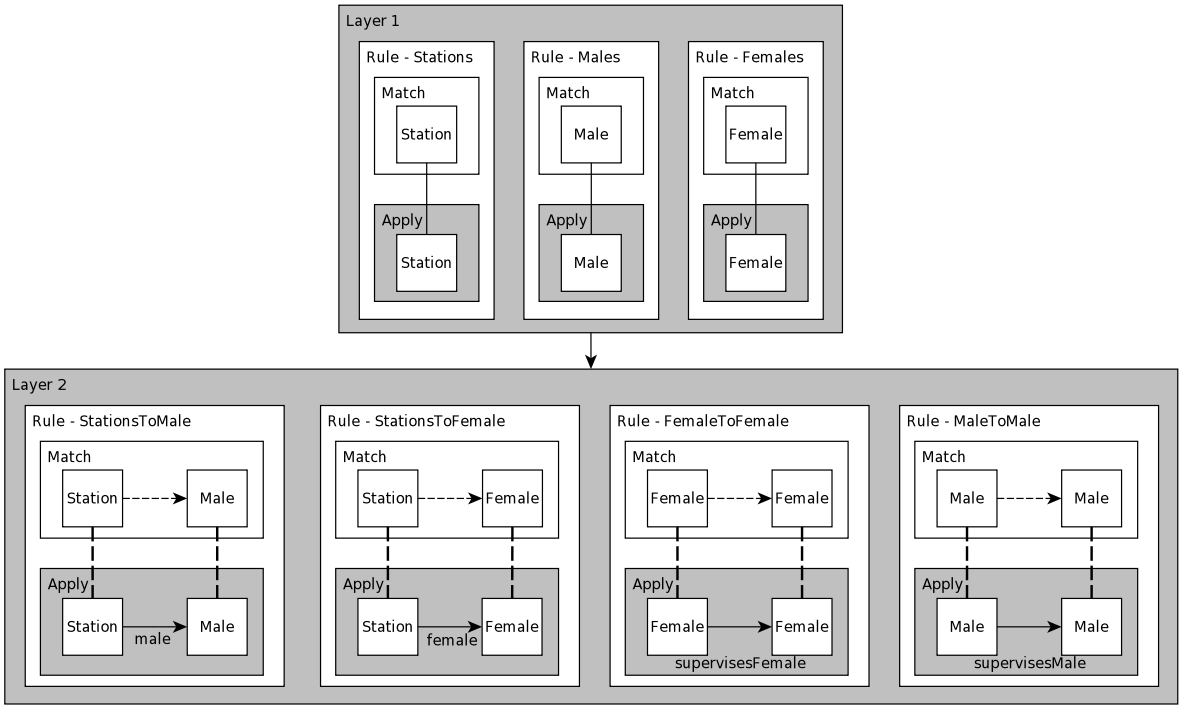
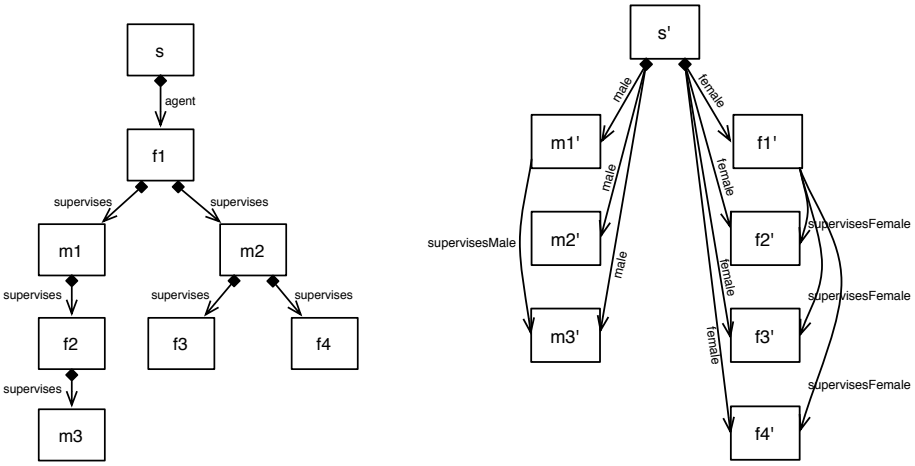


Fig. 2: The *Police Station* model transformation expressed in DSLTrans

Besides the fact that DSLTrans' transformations are free of constructs that imply unbounded recursion or non-determinism, DSLTrans' transformations are strictly outplace, 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.

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.

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.



(a) Original model (b) Transformed model

Fig. 3: Model before and after transformation

Each individual transformation rule in the transformation 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. A formal definition of a transformation rule is found in Definition 13 in Section 3.

As an 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.

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 (as formally stated in Definition 16 of model transformation, in Section 3 of this paper). This requirement is to prevent the case where a rule could not execute independently of another rule, except for the cases when such dependency is explicitly defined by backward links. This is undesirable for the algorithm as presented here as we will explain later. However, as seen in [18], the expressiveness of the transformations our algorithm can examine is not restricted. In that work, we detail an operational rule processing step to handle overlapping rules.

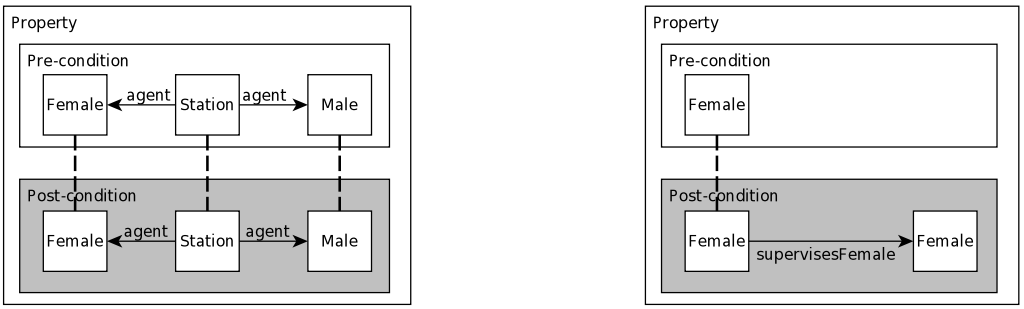
2.1 Properties to Prove

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, a 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 of the property proving process are discussed in Section 6.

As a brief example of property syntax and semantics, 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



(a) Property 1 – Expected to hold

(b) Property 2 – Not expected to hold

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

model where the pre-condition holds, but the post-condition does not. This would be an input model that contains only one female officer, as there will only be one female officer in the output model.

We shall discuss our property proving technique in Section 6. Following this, experiments in Section 8 will present experimental results from proving these two properties on the Police Station transformation.

2.2 DSLTrans Constructs

This section will describe all of the DSLTrans constructs involved in our property-proving algorithm. These constructs are found in the transformation presented in Figure 2. Formal details for the handled constructs are found in Section 3.2 and Section 3.3, while Section 3.4 briefly introduces the formal semantics of this subset of DSLTrans. The visual syntax presented here is based on the DSLTrans Eclipse plug-in syntax [8].

– **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.
  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. When used in a rule, backward links match over traceability links between elements of the transformation's input and output models. These traceability links are implicitly created when any rule is executed during the transformation. Backward links thus make it possible to refer in a rule to output elements created by a previous layer.
  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's output. Apply links will always be created in the transformation's output. 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 the formal concepts that will be used throughout all this paper. We start in Section 3.1 by a few (typed) graph concepts that will be used as mathematical building blocks throughout this paper. In particular we will introduce the notion of typed graph, typed graph union and subset, and useful relations between typed graphs based on homomorphisms. Notice that these concepts are well known from graph theory and are only slightly customized for our purposes.

Armed with the fundamental notion of typed graph, we can then introduce other formal concepts in Sections 3.2, 3.3 and 3.4 which describe the artifacts from the modeling and transformation world that we require for our verification technique. Naturally, we start by introducing the central notion of *metamodel*, allowing the description of the inputs and outputs of a model transformation. Other fundamental notions we will define in this section are *model*, *transformation rule*, *transformation* and the semantic concept of *model transformation execution*. Several auxiliary and intermediate notions for defining the syntax and semantics of our techniques will also be introduced here.

Note that this section presents a collection of formal tools that are used in the subsequent sections of this paper where the contributions of this paper are presented. It is meant as a formal reference for the upcoming formal development. This section can be safely skipped or skimmed by the reader, who can return to these definitions punctually to understand the detailed formal underpinning of our approach.

3.1 Typed Graphs

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 6-tuple $\langle V, E, (s,t), \tau, VT, ET \rangle$ *where:* V *is a finite set of vertices;* E *is a finite set of directed edges connecting the vertices* V*;* (s,t) *is a pair of functions* $s : E \to V$ *and* $t : E \to V$ *that respectively provide the source and target vertices for each edge in the graph; function* $\tau : V \cup E \to VT \cup ET$ *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 $\langle V, E, (s,t), \tau, VT, ET \rangle, \langle V', E', (s',t'), \tau', VT', ET' \rangle \in \text{TG}$ *be typed graphs, where* VT *and* ET' *are disjoint sets, as well as* VT' *and* ET*. The typed graph union is the function* $\sqcup : \text{TG} \times \text{TG} \to \text{TG}$ *defined as:*

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

For the formal development of our technique, 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 $\langle V,E,st,\tau,VT,ET\rangle = g$ *and* $\langle V',E',st',\tau',VT',ET'\rangle = g' \in \text{TG}$ *be typed graphs. A typed graph homomorphism between* g *and* g' *is a function* $f : V \to V'$ *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* $\tau(e) = \tau(e')$. *The domain of* f *is noted* $Dom(f)$ *and the co-domain of* f *is noted* $CoDom(f)$. *When an* injective *typed graph homomorphism* f *exists between* g *and* g' *we write* $g \overset{f}{\lhd} g'$, *or simply* $g \lhd g'$ *when the context is unambiguous. When a* surjective *typed graph homomorphism* f *exists between typed graphs* g *and* g' *we write* $g \overset{f}{\blacktriangleleft} g'$, *or also simply* $g \blacktriangleleft g'$ *in an unambiguous context.*

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

Notation: In order to simplify our notation, when the context is unambiguous we will abbreviate a typed graph $\langle V,E,st,\tau,VT,ET\rangle$ as as 4-tuple $\langle V,E,st,\tau\rangle$. Also, given a typed graph $g \in \text{TG}$, will use the notation $Components(g)$ to describe the set of strongly connected typed graphs in g. Finally, we will use the notation $g|_t$ to refer to the restriction of graph g to its subgraph containing only edges of type t.

3.2 Metamodel and Model-Related Constructs

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. Two metamodels, the *organization language* and *gender language* are depicted in Figure 1.

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* τ *is a bijective typing function. Additionally we also have that: if* $v \in V$ *then* $\tau(v) \in VT \times \{abstract, concrete\}$, *where* VT *is the set of vertex type names; if* $e \in E$ *then* $\tau(e) \in ET \times \{containment, reference\}$, *where* ET *is a set of edge type names. The set of all metamodels is called* META.

A formal metamodel is a particular kind of typed graph where vertices represent classes and edges represent 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 inheritance 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. For simplification purposes, we do not model association cardinalities in our formal notion of metamodel as cardinalities are not strictly necessary in our development.

Definition 7 *Expanded Metamodel*
Let $mm = \langle V,E,st,\tau,\leq\rangle \in \text{META}$ *be a metamodel. The expansion of mm, noted* $mm^\star$, *is a typed graph* $\langle V',E',st',\tau'\rangle \in \text{TG}$ *built as follows:*

- $V' = V \setminus \{v \in V \mid \tau(v) = (\cdot^1, abstract)\}$;
- $v_1 \xrightarrow{e} v_2 \in E'$ *if* $v_1 \xrightarrow{e} v_2 \in E$ *and* $\tau(v_1) = (\cdot, concrete)$ *and* $\tau(v_2) = (\cdot, concrete)$;
- *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 construct where all the relations between types of a metamodel are made explicit, rather than remaining implicit in the specialization hierarchy. It is built by adding to the original metamodel typed

[1] In our mathematical development we use a 'dot' notation to represent that we do not care about the value of a particular variable in a given context.

graph a relation of type t between two classes of the metamodel, whenever those classes specialize two classes that are also related by a relation of type t. Abstract classes and their relations do not carry over to the expanded metamodel. Expanded metamodels will be used in the subsequent text to facilitate formal the treatment of any structure involving polymorphism.

Definition 8 *Metamodel Instance*
An instance of a metamodel $mm = \langle V', E', st', \tau', \leq \rangle \in$ META *is a typed graph* $\langle V, E, st, \tau \rangle \in$ TG*, where the co-domain of* τ *equals the co-domain of* τ'*. Also, there is a typed graph homomorphism* $f : V \to V'$ *from* $\langle V, E, st, \tau \rangle$ *to the expanded metamodel* $mm^\star$ *and the graph* $\langle V, \{e \in E \mid \tau(e) = (\cdot, containment)\} \rangle$ *is acyclic. The set of all instances for a metamodel mm is called* 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 cyclic containment relations, as enforced by EMF.

Definition 9 *Containment Transitive Closure*
The containment transitive closure of a metamodel instance $\langle V', E', st', \tau' \rangle \in$ INSTANCE^{mm} *is a typed graph* $\langle V, E, st, \tau \rangle$ *where we have that* $V = V'$*,* $\tau' \supseteq \tau$ *and* τ*'s co-domain is the union of the co-domain of* τ' *and the set* $\{indirect\}$*. We also have that* $E' = E \cup E_c^*$ *where* E_c^* *is the transitive closure of the set* $\left\{ v \xrightarrow{e} v' \mid \tau(v \xrightarrow{e} v') = (\cdot, containment) \right\}$ *and if* $e \in E \setminus E'$ *then* $\tau(e) = indirect$*. We denote* mi^* *the containment closure of a metamodel instance* $mi \in$ 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 the definitions that follow we will use the $*$ notation, as in Definition 9, 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. Note that the $*$ notation is different from the $\star$ notation, introduced in Definition 7 for an expanded metamodel.

Definition 10 *Model*
A model of a metamodel $mm = \langle V', E', st', \tau', \leq \rangle \in$ META *is a metamodel instance* $\langle V, E, st, \tau \rangle \in$ INSTANCE^{mm}*, such that: there exists an injective typed graph homomorphism* $f : V \to V'$ *from* $\langle V, E, st, \tau \rangle$ *to metamodel* $mm^\star$ *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* MODEL^{mm}.

A model, as per Definition 10, 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. Two models can be observed in Figure 3, which are respectively instances of the *Organization language* and *Gender language* found in Figure 1.
Note that the containment constraint does not necessarily lead to infinite models in the case of containment relations with the same source and target classes. In fact, if the cardinality of the target class is allowed to be zero, then it is not necessary that the containment relation is instantiated. For example, this is the case for the containment relation *supervise* in the metamodel of Figure 1a.

Definition 11 *Input-Output Model*
An input-output model is a 6-tuple $\langle V, E, (s,t), \tau, Input, Output \rangle$*, where:* $Input = \langle V', E', st', \tau' \rangle \in$ INSTANCE^{sr} *is a model;* $Output = \langle V'', E'', st'', \tau'' \rangle \in$ INSTANCE^{tg} *is a metamodel instance; Input and Output are disjoint. Additionally we have that* $V = V' \cup V''$*,* $E \subseteq E' \cup E''$ *and* $\tau \subseteq \tau' \cup \tau''$*, where the co-domain of* τ *is the union of the co-domains of* τ' *and* τ'' *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* IOM$^{sr}_{tg}$.

An input-output model is an object we will use when defining the semantics of a DSLTrans model transformation in Section 3.2. It is composed of two metamodel instances, one called the *input* and the other one the *output*. An input-output model allows the representation of intermediate operational 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.

Definition 12 *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* $V = V'$, $E' \supseteq E$ *and the co-domain of* τ *is the union of the co-domains of* τ' *and the set* $\{indirect\}$. *Also, if* $e \in E \setminus E'$, *then we have that* $\tau(e) = indirect$. *Given a metamodel mm, the set of all metamodel patterns for mm is called* PATTERN^{mm}. *The set of all indirect metamodel patterns for mm is called* IPATTERN^{mm}.

Metamodel patterns are introduced in Definition 12 as an intermediate notion, formally equal to metamodel instances. An *indirect* metamodel pattern is a metamodel pattern that includes edges typed as *indirect*. Both structures will be used as building blocks in the construction of transformation-related structures in the upcoming text.

3.3 Syntactic Transformation Constructs

This section will detail the abstract syntax of the constructs involved in a DSLTrans transformation.

Definition 13 *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 an 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; Match and Apply are disjoint. We also have that* $V = V' \cup V''$, $E \subseteq E' \cup E''$ *and* $\tau \subseteq \tau' \cup \tau''$, *where the co-domain of* τ *is the union of the co-domains of* τ' *and* τ'' *and the set* $\{trace\}$. *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* $\tau(e) = trace$. *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* $\tau(e) = trace$, *or that* E'' *is not empty. The set of all transformation rules for a source metamodel sr and a target metamodel tg is called* RULE^{sr}_{tg}.

A transformation rule is the elemental block of a model transformation. Several transformation rules can be observed in the Police Station transformation in Figure 2. A formal 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 or an edge, 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 containment relations in 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 *trace*.

Definition 14 *Matcher of 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 matcher, noted* $\|rl\|$, *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* $\tau(e) = trace$ *and* $V' = V_m \cup \{ v_1 \mid v_1 \xrightarrow{e} v_2 \in E \wedge \tau(e) = trace \}$.

Definition 14 introduces the notion of matcher for a transformation rule which consists solely of the match pattern of a rule and its backward links, if any. The matcher of a rule constitutes the complete pattern that a DSLTrans rule attempts to match over a input-output model during rule execution. Traceability links between input and output model elements generated during transformation execution are matched by transformation rules' backward links, as informally explained in Section 2.2.

Definition 15 *Expanded Transformation Rule*
Let $rl = \langle V, E, st, \tau, Match, Apply \rangle \in \text{RULE}^{sr}_{tg}$ *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* $rl^\star$ *is a set of transformation rules built as follows:*

- $rl \in rl^\star$;
- $\langle V, E, st, \tau', Match, Apply \rangle \in rl^\star$ *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 such that polymorphism is correctly handled in the developments that follow.

Definition 16 *Layer, Model Transformation*
A layer is a finite set of transformation rules $l \subseteq \text{RULE}^{sr}_{tg}$.

[2] We use the simplified ε notation to denote empty n-tuples structures.

The set of all layers for a source metamodel sr and a target metamodel tg is called LAYER_{tg}^{sr}. *A model transformation is a finite list of layers denoted* $[l_1 :: l_2 :: \ldots :: l_n]$ *where* $l_k \in \text{LAYER}_{tg}^{sr}$ *and* $1 \leq k \leq n$, $n \in \mathbb{N}$. *We also impose that for any pair of rules* $rl_1, rl_2 \in \bigcup_{1 \leq k \leq n} l_k$, *if* $\|rl_1\| \cong rl$ *and* $rl \sqsubseteq \|rl_2\|$ *then* rl_2 *appears in a layer later than* rl_1 *and the apply parts of* rl_1 *and* rl_2 *are not isomorphic. The set of all transformations for a source metamodel s and a target metamodel t is called* TRANSF_{tr}^{sr}.

Definition 16 formalises the abstract syntax of a model transformation, introduced in Section 2. An example of a model transformation can be observed in Figure 2, the Police Station transformation. 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 16 imposes that, for any two pair of rules in the transformation, the matcher of the second rule never partially or totally subsumes (or contains) the matcher of the first rule, unless the second rule is in a subsequent layer and produces something more than the first rule. This condition avoids situations where the execution of a rule in a DSLTrans model transformation necessarily implies the execution of another rule (except for when rules having backward links necessarily execute because all their dependencies were created during the execution of previous layers).

Notation: We naturally extend to transformation rules (Definition 13) and transformation executions (Definition 17) the typed graph operations introduced in Section 3. Also, given a structure such as transformation rule $rl = \langle V, E, st, \tau, Match, Apply \rangle$, we will refer to the structure's components by using the component's name followed by the variable that holds the structure in between parenthesis. For example, we will write $V(rl)$ to designate the V component of rl or $Apply(rl)$ to designate rl's $Apply$ component.

3.4 Semantic Transformation Constructs

In the definition that follows we introduce the notion of execution of a DSLTrans model transformation. For our purposes it is sufficient to introduce it as an input-output model (see Definition 11), containing the input model for the transformation, the produced output, and the traceability links built during execution. Due to space limitations, we cannot introduce the semantics of DSLTrans in the main text of this paper. We thus refer the reader to Section 3.2 for a formal description on how DSLTrans transformation executions are built.

Definition 17 *Model Transformation Execution*
Let $tr \in \text{TRANSF}_{tg}^{sr}$ *be a transformation and* $input \in \text{MODEL}^{sr}$ *be a model. Assume we also have that:*

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

A model transformation execution is the input-output model $\langle V', E', st', \tau', input, output \rangle \in \text{IOM}_{tg}^{sr}$, *where* $output \in \text{IOM}_{tg}^{sr}$ *is an input-output model. The set of all model transformation executions for transformation tr is written* $\text{EXEC}(tr)$. *A model transformation with an empty input model is noted* ε_{ex}. *Note that relation trstep is formally defined in Appendix B.*

Finally, as stated in Definition 17, we consider a model transformation execution to be the input-output model (IOM) resulting from executing a set of rules on a starting IOM. This starting IOM includes the transformation's input in its input part and has an empty output part. The starting IOM represents the first step of the transformation when no rule has been executed yet. A transformation execution results from executing all the rules in a DSLTrans model transformation.

4 Building Path Conditions

This section will present how path conditions are structured to represent symbolic rule execution. As well, we present our approach to building a set of path conditions to represent all executions of a DSLTrans transformation.

4.1 Symbolic Execution

Our algorithm operates on the principle of symbolic execution to build up these path conditions. 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*" [19]. 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. These control and/or data dependencies must be taken into consideration during path condition construction.

As in program symbolic execution, each path condition in our approach *abstracts* as many concrete executions as there are input/output models that satisfy them. This is formulated as an *abstraction relation* as further explained in Section 5.

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

In order to present the intuition of path conditions and symbolic executions, we first discuss the idea of *rule combinations*.

As seen in Section 2, a layer in a DSLTrans transformation contains a number of rules. We can create a set of rule combinations for this layer by taking the powerset of all rules in that layer. Each rule combination in this set will 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.

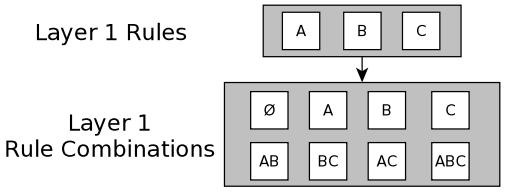


Fig. 5: Rule combinations created for a transformation layer

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. This abstraction is key to our approach, as it allows us to create a finite set of path conditions to abstract over an infinite set of transformation executions, as seen in Section 5.

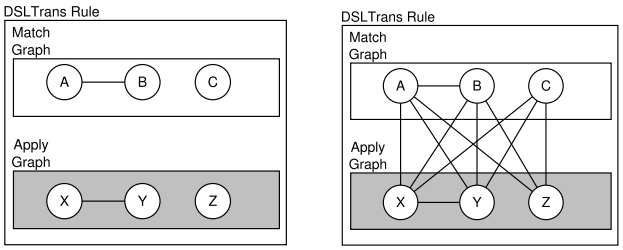
We also note that rule *combinations* are created, and not rule *permutations*. This follows from the semantics of DSLTrans as described in Section 2, as transformation rules in a layer will execute in a non-deterministic order but produce a deterministic result, by construction of the semantics of DSLTrans. As a final note, the transformation executions that these rule combinations represent always terminate, also by construction of the semantics of DSLTrans [6].

We base our concept of path conditions on these rule combinations. However, as DSLTrans allows for dependencies between rules, we cannot create path conditions for the transformation by taking the powerset of all rules. Instead, our approach must move layer-by-layer and resolve the dependencies between rules. The next two sections will introduce the concepts of traceability and dependency, before we briefly discuss the syntax and semantics of path conditions themselves.

4.2.1 Traceability DSLTrans rules allow for dependencies to be specified on which elements of the output model were created from specific elements of the input model. To resolve these dependencies, traceability information for the transformation is created during the execution of a DSLTrans model transformation [6]. In our verification approach, we store this same information as symbolic *traceability links*, in order to record which elements belong to the same DSLTrans rule.

At a particular point in the path condition construction process, symbolic 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 symbolic traceability link is built between the two if the apply element is not connected to a backward link (as explained below). 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 symbolic traceability link creation process is shown in Figure 6. Note that symbolic traceability links are a solid line between match and apply elements in our visual notation.

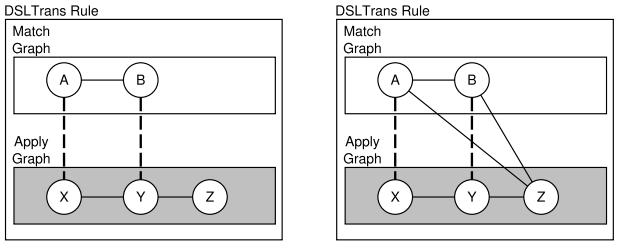


(a) Before symbolic traceability links added (b) After symbolic traceability links added

Fig. 6: Symbolic traceability links created for an abstract DSLTrans rule

4.2.2 Backward Links The dependencies in a DSLTrans rule are specified using the *backward link* construct, as further detailed in Section 2.2 and Definition 13. Section 4.4.2 will discuss how these dependencies are then resolved during our symbolic execution approach.

Figure 7a demonstrates how backward links are used within a rule. The rule shown contains a backward link, which defines the dependency that an element of type X was created from an element of type A, and an element of type Y was created from an element of type B. If this dependency is satisfied, then another element of type Z should be created. This element should be associated with the Y element.



(a) Rule with backward links (dashed lines) (b) Traceability links added

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

Figure 7b shows the rule after symbolic traceability links have been added. Two symbolic traceability links are created from the Z element to the A and B elements in the match graph to store traceability information. Note that no symbolic traceability links are built between two elements connected by backward links, as these links have already been built in a previous layer.

4.2.3 Syntax and Semantics A path condition represents the symbolic execution of a set of DSLTrans rules, similar to a rule combination as explained above. Again, we use an abstraction over the number of times a rule has symbolically executed. Each path condition will represent that a rule has not executed, or has executed one or more times.

The path condition generation algorithm will symbolically combine transformation rules into a path condition. Each path condition will then abstract a set of concrete transformation executions, as defined by our abstraction relation in Section 5.

As seen in the rest of this section, the structure of path conditions is similar to that of DSLTrans rules. The match graph of a path condition represents a pattern that must be present in the input model of the transformation, while the apply graph is a pattern which will be instantiated in the output model of the transformation. Symbolic traceability links are also kept between elements in the match and apply graphs to retain traceability information.

The formal definition of a path condition is presented in Definition 18.

Definition 18 *Path Condition*
A path condition is a 7-tuple $\langle V, E, (s,t), \tau, Match, Apply, Rulecop \rangle$*, where:* $Match = \langle V', E', st', \tau' \rangle \in$ IPATTERN^{sr} *is an indirect pattern;* $Apply = \langle V'', E'', st'', \tau'' \rangle \in$ PATTERN^{tg} *is a pattern; Match and Apply are disjoint graphs. We also have that* $V = V' \cup V''$, $E \subseteq E' \cup E''$ *and* $\tau \subseteq \tau' \cup \tau''$ *where the co-domain of* τ *is the union of the co-domains of* τ' *and* τ'' *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 Rulecop component in the 7-tuple contains the set of rule copies used in the construction of the path condition, where each rule copy 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* PATHCOND$^{sr}_{tg}$ *and the empty path condition is noted* ε_{pc}*.*

Similarly to a transformation rule (see Definition 13), 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 17). The path condition structure also contains a *Rulcop* set that allows identifying individually all the copies of rules that were used when building the path condition's typed graph. Note that we refer to *copies* of rules as,

despite the fact that a path condition normally only contains one copy of each rule, in certain situations a rule may be used multiple times in the construction of a path condition. This will be explained further ahead in this section.

Notation: Given a path condition $pc = \langle V, E, st, \tau, Match, Apply, Rulecop \rangle \in \text{PATHCOND}^{sr}_{tg}$ we refer to the set of transformation rules in pc identified by the $Rulecop$ relation as $Rulecop(pc)$. Also, because a path condition is a particular kind of a typed graph, we naturally extend the basic notation of operators and homomorphisms on typed graphs defined in Section 3 to path conditions.

4.3 Path Condition Generation Algorithm

This section will describe how path conditions are constructed for a DSLTrans 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 have been 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 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 symbolically executed, thereby abstracting a set of transformation executions through our abstraction relation. Combining a path condition with a rule will produce one or more path conditions depending on how the rule combines with the rules already represented by the path condition. The pre- and post- conditions defined by the path condition will be modified according to the elements found in that rule.

Each of the new path conditions created from combining a rule with a path condition will then be combined with the next rule in the layer. A small example is shown in Figure 8b, where a path condition is combined with two rules. Note that a rule can combine with a path condition in multiple ways (differentiated by prime marks in the figure). Figure 9 shows how path conditions from the previous layer are sequentially combined with all the rules from the current layer. All the path conditions for the layer are then collected to produce the final working set of path conditions for the layer.

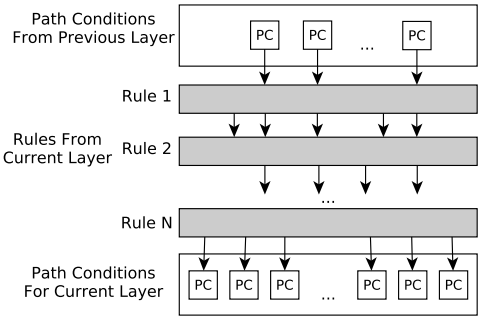


Fig. 9: Creating all path conditions for a layer

4.4 Combining a Path Condition with a Rule

We will now examine the combination step between one path condition and one rule, which produces a set of new path conditions. A formal and generic definition of this step will be presented first, before we explain the specialized combination possibilities with figures and informal text.

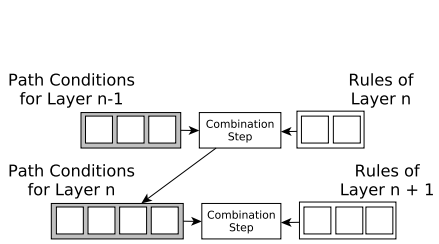
Definition 19 *Combination of a Path Condition with a Rule*

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

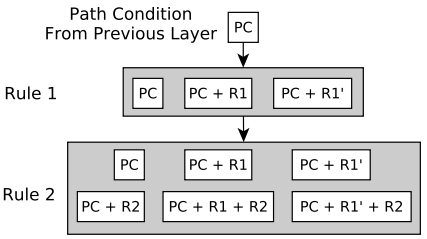
$$pc \overset{trace}{\sqcup} rl = \langle V, E, st, \tau, Match, Apply, Rulecop \rangle$$

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* $v_1 \in Apply(V'')$, $v_1 \notin Apply(V')$, $v_2 \in Match(V'')$ *and also that* $\tau'(e) = trace$. *Additionally,* $Match = Match' \sqcup Match''$ *and* $Apply = Apply' \sqcup Apply''$. *Finally, we have that:* $Rulecop = Rulecop' \cup rl$.

Definition 19 shows the formal definition of combining a path condition with a rule. When a path condition is combined with a rule their typed graphs are united. Additionally, symbolic traceability links will be built at this time between the newly added apply elements of the rule and all of the



(a) Previous path conditions are combined with rules



(b) Combining a path condition with two rules

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

rule's match elements. As a reminder, the link creation algorithm and examples have been introduced in Section 4.2.1. 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 links shared by the path condition and the rule graph. In the mathematical development that follows we will often refer to the joint parts of two or more typed graphs using the term "glue".

We will now discuss the combination step possibilities. 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 4.2.1, 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.

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

4.4.1 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

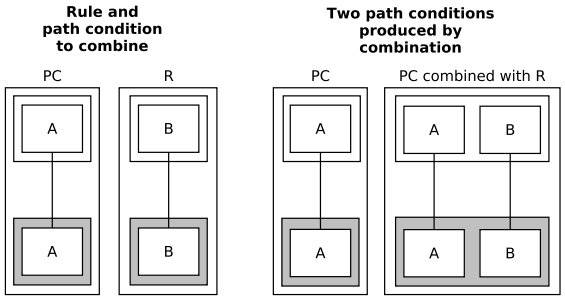


Fig. 10: R has no dependencies

a specific input model, and thus R would not execute in these 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 20.

Definition 20 *Path Condition and Rule Combination – No Dependencies*
The combination of a path condition pc and a rule rl, when rl has no dependencies, is described by the relation $\overset{combine}{\rightarrow} \subseteq \text{PATHCOND}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{RULE}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$, *defined as follows:*

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

Relation $\overset{combine}{\rightarrow}$ in Definition 20 models the operational combination step shown in Figure 8b (the vertical black arrows between boxes). The relation has three input arguments: the first argument is the original path condition from the pre-

vious layer (shown as the topmost box in Figure 8b with label *PC*); the second argument is the set of path conditions accumulated thus far by combining other rules in the current layer with the original path condition; and the third argument is the rule from the current layer now being combined. The fourth argument of the relation, the relation's output, is the new set of path conditions resulting from this combination.Briefly, the equation in Definition 20 states that whenever a rule has no backward links typed as *trace* (i.e. no dependencies), all path conditions in the accumulator set are kept, along with the result of combining all the path conditions in the accumulator set with the current rule.

4.4.2 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 symbolic traceability links in PC. Note that symbolic 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 symbolic traceability links in PC, then in the transformation executions abstracted by PC, R cannot execute. Again, PC will be copied unchanged to the new set of path conditions, but no new path condition will be created. This case is shown in Figure 11, where the backward links between the two B elements in R cannot match over the symbolic traceability link in PC. Definition 21 describes this case formally.

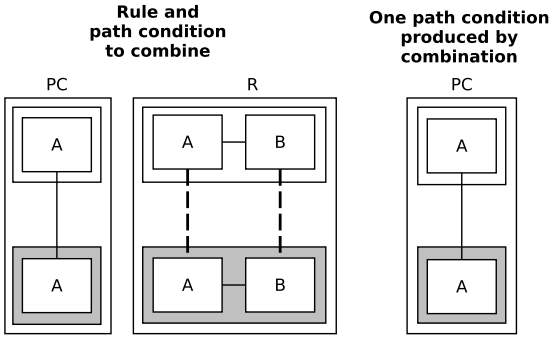


Fig. 11: R's dependencies are not satisfied by PC

Definition 21 *Path Condition and Rule Combination – Unsatisfied Dependencies*
The combination of a path condition pc and a rule rl, when rl has dependencies that are not satisfied by pc, is described by the relation $\overset{combine}{\rightarrow} \subseteq \text{PATHCOND}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{RULE}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$*, defined as follows:*

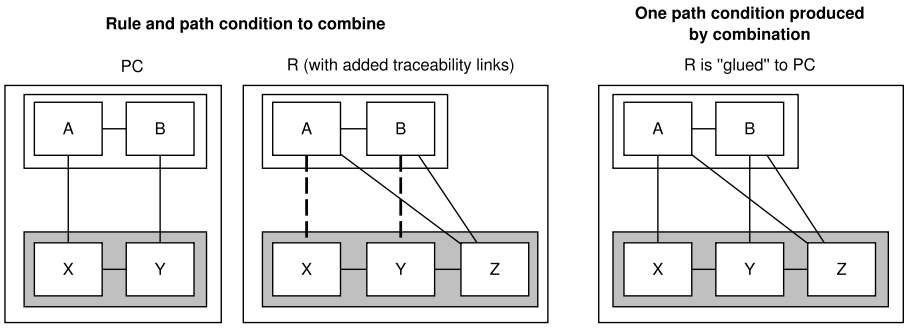
$$\frac{\neg\left(rl|_{trace} \blacktriangleleft pc|_{trace}\right)}{\langle pc, AC, rl\rangle \xrightarrow{combine} AC}$$

According to the pre-conditions of the equation presented in Definition 21, a path condition does not satisfy the dependencies present in a rule if there is no surjective typed graph homomorphism between the backward links of the rule and the symbolic traceability links of the path condition. Besides expressing the fact that all backward links must exist as symbolic traceability links the path condition, the surjective homomorphism allows modeling the case where dependencies expressed by two (or more) backward links between similarly typed elements can be satisfied one single symbolic traceability link in the path condition. This is the case, for example, of rule *FemaleToFemale* in the *Police Station* in Figure 2. The two similarly typed backward links in this rule are satisfied by a path condition containing only the rule *females* generated from the first layer of the transformation, holding one single symbolic traceability link.

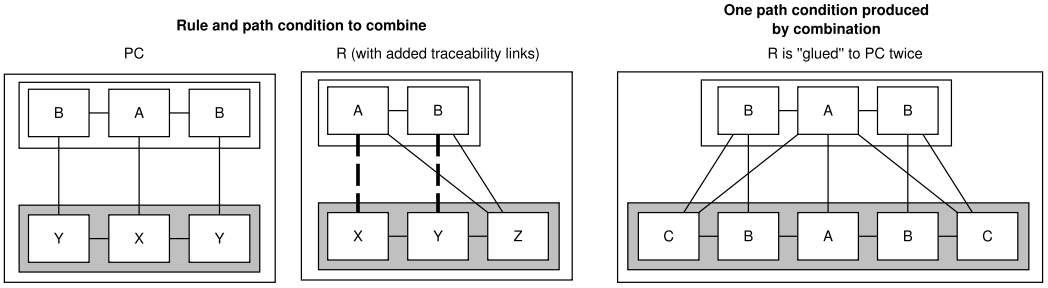
Partially- and Totally- Satisfied Dependencies Consider the possibility that the backward links of R can be found in PC, and R's dependencies are met. The question then becomes whether the rule R **may** or **will** execute in the abstracted transformation executions.

To resolve this question, 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', where R **will** necessarily execute in the abstracted transformation executions. Otherwise, we denote the 'partially-satisfied' case, where R **may** execute. Note that we break up these cases for ease of explanation only. Formally, both cases are encompassed by Definition 24.

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. This



(a) Totally satisfied at one location



(b) Totally satisfied at multiple locations

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

"gluing" operation is also defined formally in Definition 24, as the addition of a delta graph.

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 that the input model for the transformation does not have this association present. In these transformation executions 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 does not symbolically execute. The second is where R symbolically executes at the matched location. Therefore, R is glued onto PC, with the gluing step the same as in the totally-satisfied case above.

Note that this gluing procedure must consider all matching possibilities, for each location the rule might match over the input model. For example, in Figure 14, rule R has a backward link that can be partially matched on two locations in PC: the left-hand and right-hand pairs of traceability links. Therefore, there are four possibilities for how R would match over PC: not at all, on the left-hand side of PC, on the right-hand side, 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 left-hand side, to add the elements that do not exist in PC already. The third is where the gluing will occur on the right-hand side. 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.

As we have done for the previous cases, let us now formally define the combination step when a rule has partially and/or totally defined dependencies. As these cases are more complex than the previous two, we will need to construct the mathematical model of this case incrementally. We will start by an auxiliary relation that partially or totally combines a set of path conditions with a rule.

Definition 22 *Single Partial and Total Combination of a Set of Path Conditions with a Rule*
The single rule partial and total combination relations $\overset{p\_comb}{\rightarrow}$

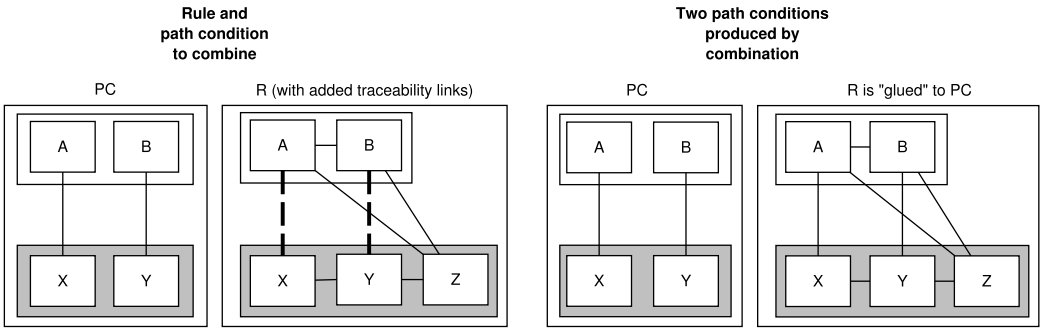


Fig. 13: R's dependencies are partially satisfied by PC

and $\overset{t\_comb}{\rightarrow}$*, both having having signature* $\mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{RULE}^{sr}_{tg} \times \text{RULE}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$ *are defined as follows:*

$$\frac{rl \cong rl_{glue} \sqcup ma_{\Delta}}{\langle AC, rl, rl_{glue} \rangle \xrightarrow{p\_comb} AC \ \cup \ \bigcup_{pc \in AC} pc \overset{trace}{\sqcup} (rl_{glue} \sqcup ma_{\Delta})} \quad (1)$$

$$\frac{rl \cong rl_{glue} \sqcup ma_{\Delta}}{\langle AC, rl, rl_{glue} \rangle \xrightarrow{t\_comb} \bigcup_{pc \in AC} pc \overset{trace}{\sqcup} (rl_{glue} \sqcup ma_{\Delta})} \quad (2)$$

Let us start by introducing relation $\overset{p\_comb}{\rightarrow}$, presented in Equation (1) of Definition 22. The relation takes as arguments a set of path conditions being accumulated for the current layer, the rule to be combined, and an rl_{glue} argument indicating the place in each of the input path conditions the rule should be anchored to during the combination step. The relation's output is a new set of path conditions. This new set includes all the original path conditions, as well as each path condition in the accumulator set "glued" to a copy of rule being examined. Note that the relation $\overset{t\_comb}{\rightarrow}$ in Equation (2) of Definition 22 is similarly defined, except for the fact path conditions in the accumulator set are not preserved in the relation's output set.

Let us now define how a rule is combined with a path condition, whenever its backward links can be found several times in that path condition. This situation is described in the examples in Figure 12b and Figure 14. We formalize it in Definition 23, by means of relations $\overset{p\_step}{\rightarrow}$ and $\overset{t\_step}{\rightarrow}$. These two relations operationally describe the sequence of steps necessary to "glue" a rule at multiples places of a path condition. The set of places targeted in the path condition for receiving a copy of the rule is given by the sets *partialSet* and *totalSet* (found respectively in Equation (2) and Equation (4) of Definition 23). As expected, these sets contain the set of traceability links in the path condition where copies of the rule need to be anchored to.

Definition 23 *Multiple Partial and Total Combination of a Set of Path Conditions with a Rule*

The multiple rule partial and total combination relations $\overset{p\_comb}{\rightarrow}$ *and* $\overset{t\_comb}{\rightarrow}$*, both having having signature* $\mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \mathcal{P}(\text{RULE}^{sr}_{tg}) \times \text{RULE}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$ *are defined as follows:*

$$\frac{}{\langle AC, rl, \emptyset \rangle \xrightarrow{p\_step} AC} \quad (1)$$

$$\frac{rl_{glue} \in partialSet, \ \langle AC, rl, rl_{glue} \rangle \xrightarrow{p\_comb} AC'', \ \langle AC'', rl, partialSet \setminus \{rl_{glue}\} \rangle \xrightarrow{p\_step} AC'}{\langle AC, rl, partialSet \rangle \xrightarrow{p\_step} AC'} \quad (2)$$

$$\frac{}{\langle AC, rl, \emptyset \rangle \xrightarrow{t\_step} AC} \quad (3)$$

$$\frac{rl_{glue} \in totalSet \ , \ \langle AC, rl, rl_{glue} \rangle \xrightarrow{t\_comb} AC'', \ \langle AC'', rl, totalSet \setminus \{rl_{glue}\} \rangle \xrightarrow{t\_step} AC'}{\langle AC, rl, totalSet \rangle \xrightarrow{t\_step} AC''} \quad (4)$$

Having Definition 22 and Definition 23 in mind, we can now proceed to define the complete combination relation of a rule with a path condition in the case of partially and totally satisfied dependencies.

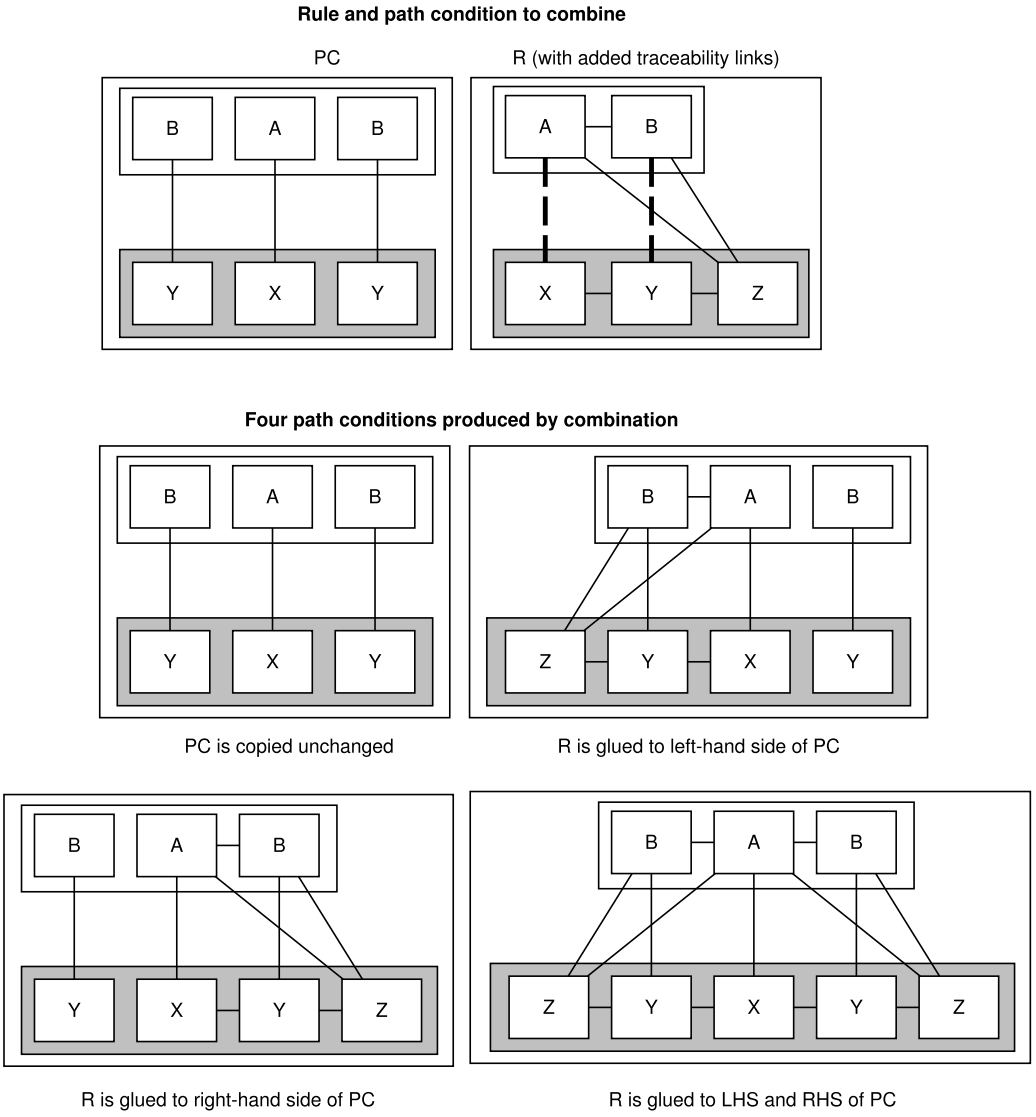


Fig. 14: R's dependencies are partially satisfied by PC, and are glued at all possible matches

Definition 24 *Path Condition and Rule Combination – Partially and Totally Satisfied Dependencies*

The combination of a path condition pc and a rule rl, when rl has dependencies that are satisfied by pc, is described by the relation $\overset{combine}{\rightarrow} \subseteq \textsc{PathCond}^{sr}_{tg} \times \mathcal{P}(\textsc{PathCond}^{sr}_{tg}) \times \textsc{Rule}^{sr}_{tg} \times \mathcal{P}(\textsc{PathCond}^{sr}_{tg})$*, defined as follows:*

$$\frac{\begin{array}{l} rl|_{trace} \blacktriangleleft pc|_{trace}\ , \\ \quad \langle AC, rl, partialsat(rl,pc)\rangle \xrightarrow{p\_comb} AC''\ , \\ \quad \langle AC'', rl, totalsat(rl,pc)\rangle \xrightarrow{t\_comb} AC' \end{array}}{\langle pc, AC, rl\rangle \xrightarrow{combine} AC'}$$

where

$$rl_{glue} \in partialsat(rl,pc) \iff rl_{glue} \sqsubseteq pc^* \wedge rl|_{trace} \blacktriangleleft rl_{glue} \quad \wedge \quad \nexists rl' \,.\, (rl_{glue} \sqsubseteq rl' \sqsubseteq pc^* \wedge \|rl\| \blacktriangleleft rl')$$

and

$$rl_{glue} \in totalsat(rl,pc) \iff rl_{glue} \sqsubseteq pc^* \wedge \|rl\| \blacktriangleleft rl_{glue}$$

The top equation in Definition 24 defines the $\overset{combine}{\rightarrow}$ relation for when rule *rl* has dependencies that are satisfied by path condition *pc*. The pre-conditions in the equation state that the backward links in the rule are found in the path condi-

tion, as expected. Additionally, two sequential steps perform the gluing of the rule *rl* on all path conditions in accumulator *AC*, wherever the rule is partially and/or totally found in each of those path conditions. Relations $\overset{p\_comb}{\rightarrow}$ and $\overset{t\_comb}{\rightarrow}$ presented in Definition 23 are used to model these two operational "gluing" steps. Functions $partialsat$ and $totalsat$, described in the latter part of Definition 24, are used to gather the places of path condition *pc* where copies of the rule need to be anchored to.

4.4.3 Considering Further Rules Thus far we have described how to create 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 in Definition 24 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 (variable *pc* in the main equation of Definition 24), not the specific path condition the rule is to be combined with in the accumulator argument of the $\overset{combine}{\rightarrow}$ relation. 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. This process is depicted in Figure 9 and formalized in Definition 25 by the layer combination relation $\overset{combpclayer}{\rightarrow}$.

Definition 25 *Combining a Path Condition with a Layer*
The layer combination relation $\overset{combpclayer}{\rightarrow} \subseteq \text{PATHCOND}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{LAYER}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$ *relation is defined as follows:*

$$\frac{}{\langle pc, AC, \emptyset \rangle \xrightarrow{combpclayer} AC}$$

$$\frac{rl \in layer,\ \langle pc, AC, rl \rangle \xrightarrow{combine} AC'',\ \langle pc, AC'', layer \setminus \{rl\} \rangle \xrightarrow{combpclayer} AC'}{\langle pc, AC, layer \rangle \xrightarrow{combpclayer} AC''}$$

After the step in Definition 25 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 process is modeled by relation $\overset{combpcsetlayer}{\rightarrow}$ in Definition 26.

Definition 26 *Combining a Set of Path Conditions with a Layer*
The path condition layer step relation $\overset{combpcsetlayer}{\rightarrow} \subseteq \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{LAYER}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$ *relation is defined as follows:*

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

$$\frac{pc \in AC,\ \langle pc, \{pc\}, layer \rangle \xrightarrow{combpclayer} AC',\ \langle AC \setminus \{pc\}, layer \rangle \xrightarrow{combpcsetlayer} AC''}{\langle AC, layer \rangle \xrightarrow{combpcsetlayer} AC' \cup AC''}$$

This working set of path conditions obtained for each layer 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 and is formally described in Definition 27.
After all layers have been processed, the working set of the last layer contains all the possible path conditions of the transformation. Through our abstraction relation defined in Section 5, the final set of created path conditions will represent every feasible transformation execution. Section 6 will discuss how our algorithm proves properties on these path conditions, and thus on all executions of thetransformation.

Definition 27 *Path Condition Generation*
Let $[layer :: tr] \in \text{TRANSF}^{sr}_{tg}$ *be a transformation, where* $layer \in \text{LAYER}^{sr}_{tg}$ *is a Layer and tr also a transformation. The* $\overset{pathcondgen}{\rightarrow} \subseteq \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{TRANSF}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$ *is defined as follows:*

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

$$\frac{\langle \varepsilon_{pc}, \{\varepsilon_{pc}\}, layer^{\star} \rangle \xrightarrow{layercomb} AC'',\ \langle AC'', tr \rangle \xrightarrow{pathcondgen} AC'}{\langle \varepsilon_{pc}, [layer :: tr] \rangle \xrightarrow{pathcondgen} AC}$$

$$where\ layer^{\star} = \bigcup_{rl \in l} rl^{\star}$$

Note that in Definition 27, the recursive rule considers the expansion ($layer^{\star}$) of all the rules in a layer (see Definition 15). This allows us to deal with polymorphism during path condition generation. In particular, given one rule *rl* of *layer*, we consider for path condition generation all rules containing possible of replacements of each match element in *rl* of certain type by an element belonging to one of the type's subtypes, as defined in the source metamodel *sr*.
After all layers have been processed, the working set of the

last layer contains all the possible path conditions of the transformation. Through our abstraction relation in Definition C.11, the final set of created path conditions will represent every feasible transformation execution.

Notation: We will use the abbreviation $\text{PATHCOND}(tr)$ to represent the set of path conditions AC produced for a transformation tr, where $\langle \varepsilon_{pc}, tr \rangle \xrightarrow{pathcondgen} AC$.

5 Abstraction Relation between Path Conditions and Transformation Executions

In this section we define the abstraction relation between the execution of a DSLTrans transformation and the path condition that represents it. This abstraction relation allows us to prove properties on a finite set of representative path conditions, as created by the path condition generation algorithm. As this set is finite, our technique is guaranteed to be decidable.

This section also presents our arguments 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.

Let us start by formally defining the notion of abstraction of a transformation execution by a path condition.

Definition 28 *Abstraction of a Transformation Execution by a Path Condition*
Let $tr \in \text{TRANSF}^{sr}_{tg}$ *be a DSLTrans transformation. Let also* $pc = \langle V, E, st, \tau, Match, Apply, Rules \rangle \in \text{PATHCOND}(tr)$ *of be a path condition of* tr *and* $ex = \langle V', E', st', \tau', Input, Output \rangle \in Exec(tr)$ *be an execution of* tr*. We have that* ex *is abstracted by* pc*, noted* $ex \Vdash pc$*, if and only if the set of transformation rules of* tr *combined in* pc *and the set of transformation rules of* tr *used to built* ex *is the same, and:*

$$\big(\forall rl \in Rules \,.\, Match(rl) \lhd Input^*\big) \;\wedge\; Output \blacktriangleleft Apply \tag{1}$$

and

$$\big(\forall trc \in Components(pc|_{trace}) \,.\, trc \lhd ex\big) \;\wedge\; \big(\forall e' \in E' \,\exists e \in E \,.\, \tau'(e') = trace \Longrightarrow \tau(e) = trace\big) \tag{2}$$

To understand the abstraction relation in Definition 28, recall that during the construction of a transformation execution rules are matched injectively in the input model. This information is present in the first condition of the abstraction relation (Proposition 1) via the injective typed graph homomorphism between the match part of the copies of rules "glued" onto the path condition and the containment transitive closure of the input part of the transformation execution. This relation enforces the fact that certain parts of the execution were found, or matched, by certain parts of the path condition. On the other hand, the surjection from the output of the execution towards the apply part of the path condition models the fact that the output of the execution has been completely built by instantiating the apply parts of the rules contained in the path condition.

The second condition of the abstraction relation (Proposition 2) checks for the fact that symbolic traceability links in the path condition and traceability links in the execution correctly correspond to each other. This is modeled by the fact that all strongly connected components in the path condition, composed only of symbolic traceability links, are injectively found on the execution. This injection models the fact that traceability graphs between individual or combined rules in the path condition are necessarily found in the execution. Note that components of the path condition are considered because of the fact that disconnected rules in the path condition may have matched over common elements of a particular execution. As such, a full injection between the complete traceability graph in the path condition and the execution would be incorrect. Additionally, in the second part of Proposition 2 we check the fact that every traceability link in the execution can be found in the path condition. This additional sanity check enforces that no spurious traceability links that could not have been created by the rules present in the path condition exist in the transformation execution.

It is important to mention that another abstraction relation, weaker or stronger, could have been chosen. The abstraction relation presented in Definition 28 suits our needs in the sense that it allows us to demonstrate the validity and completeness of our proof technique, as we will show in the text follows. Additionally, it is particularly interesting because it makes sure that, given a DSLTrans transformation, each of its transformation executions is abstracted by one and only one of its path conditions. This result adds to the consistency of our theory and is also exposed later in this section.

5.1 Examples

In this section, we provide a number of examples to demonstrate the workings of the abstraction relation we chose to use. Figure 15 presents the legend for the following figures.

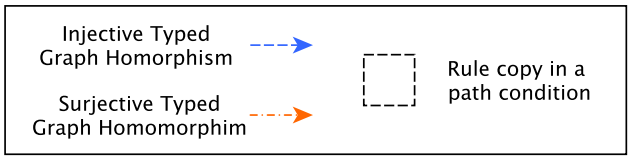
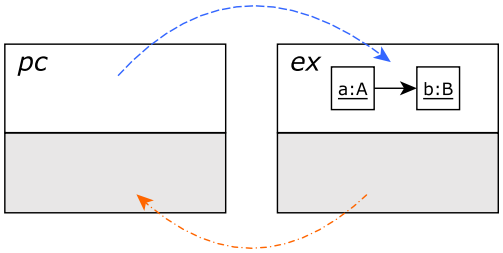
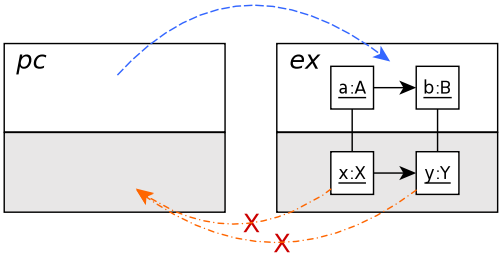


Fig. 15: Legend for abstraction relation figures

5.1.1 Example 1 – Empty Path Condition We begin by defining which transformation executions an empty path condition will abstract. Figure 16 demonstrates two cases. In each, the path condition is on the left-hand side, and a transformation execution is on the right-hand side. Note that in Figure 16a, the path condition abstracts the transformation execution, while in Figure 16b, the abstraction relation does not hold.



(a) Abstraction holds



(b) Abstraction does not hold

Fig. 16: Abstraction of transformation executions by the empty path condition

The match part of the path condition represents the preconditions 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 a typed graph injective homomorphism can be found between the match graph of the path condition, and an transformation executions. Note that in both Figure 16a and Figure 16b, an empty typed graph homomorphism satisfies this condition, highlighted by blue arrows.

The second condition for the abstraction relation is whether a typed graph surjective homomorphism can be found from the transformation execution's output model to the apply graph of the path condition. This is represented by orange arrows in Figure 16a and Figure 16b. This relation 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 an empty surjective typed graph homomorphism can be found between the output model of the transformation execution in Figure 16a 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.

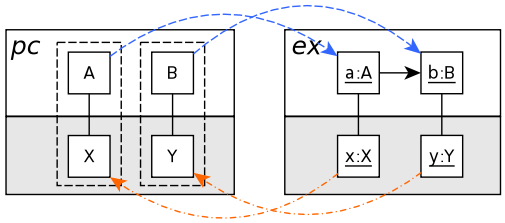
In contrast, there is no surjection between the elements of the output model in Figure 16b and the path condition. Note that the transformation execution has elements in the output model and thus at least one rule must have executed. However, the path condition does not represent that a rule has executed. Therefore, the path condition shown does not represent this execution.

5.1.2 Example 2 – Non-overlapping Rule Components This second example shows the abstraction relation between path conditions and transformation executions, when no match element of the same type appears in multiple rule components.

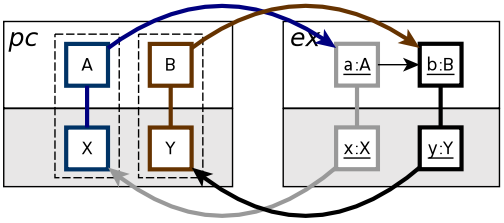
For these examples, we will represent the abstraction relation with two figures. The first will demonstrate the matching performed on match and apply graphs, while the second figure focuses on traceability link matching.

Let us first examine how the injection operates between the match elements in the path condition and the transforma-

tion executions in Figure 17a and Figure 18a. Note that this injection can be found in both cases.

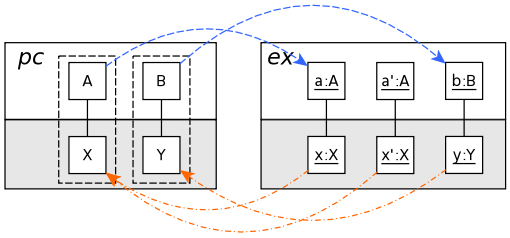


(a) Abstraction holds on match and apply graphs

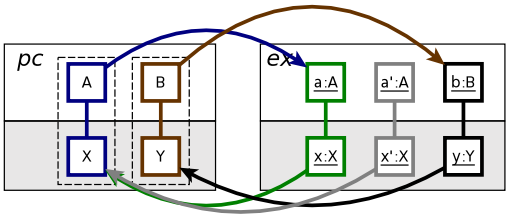


(b) Abstraction holds on traceability links

Fig. 17: Abstraction of transformation execution by non-overlapping rule components



(a) Abstraction holds on match and apply graphs



(b) Abstraction holds on traceability links

Fig. 18: Example of abstraction over multiple rule execution

Similarly, there is a surjection 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 18a, where examination of the transformation execution shows that one rule has executed twice. As mentioned before, 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.

We now examine Figure 17b and Figure 18b to resolve whether the traceability links in the path condition can be found in the transformation execution. This matching is represented by the arrows from each component highlighted in a bold outline and differentiated by colour. We note that each component in the path condition can be successfully found in the transformation execution.

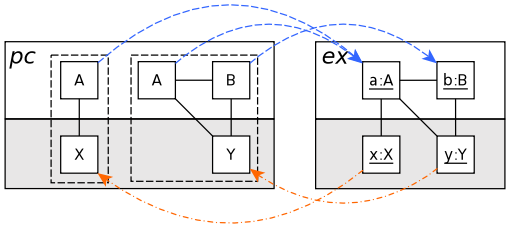
As well, there is a matching step from each individual traceability link in the transformation execution onto the path condition. Similar to the matching from the path condition, the bold components in the transformation execution figure are matched onto the path condition. We note that this matching is successful as well.

5.1.3 Example 3 – Overlapping Rule Components For these examples, the path conditions contain overlapping rule components, i.e. separate rules share match elements of the same type. Our goal is to illustrate the interaction of rule elements, where the elements of non-dependent rules may match over the same or different elements in the transformation execution.

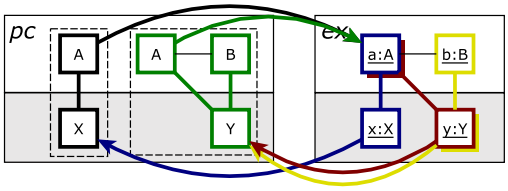
For example, the two rule components in Figure 19a correctly match over the transformation execution shown. The abstraction relation holds due to the fact that, while match elements of the same component need to be found injectively in the execution, the injection constraint does not span multiple components. This allows the match elements from different rules to match to the same input model element.

As well, Figure 19b shows the mapping from the path condition to the transformation execution. However, note that the pattern composed of the A, B, and Y elements, along with the traceability links, is to be matched as a whole. This is to ensure that the traceability links are found in the proper configuration in the transformation execution.

We also match the traceability links from the transformation execution back onto the path condition. Again, this is to ensure that no traceability links are found in the transformation execution that have not been represented in the path condition. Three matches are performed in this step, denoted by the three arrows in the bottom of Figure 19b. Each match



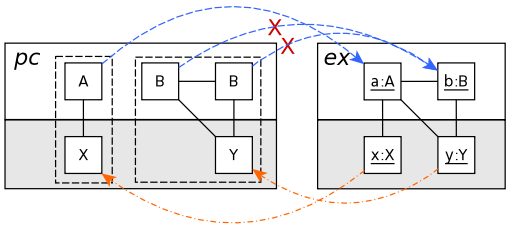
(a) Abstraction holds on match and apply graphs



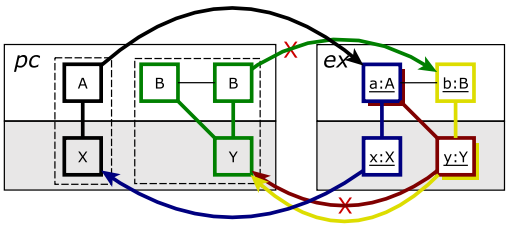
(b) Abstraction holds on traceability links

Fig. 19: Abstraction of transformation execution by overlapping rule components

is composed of a traceability link as well as immediately connected elements.



(a) Elements cannot overlap within a component

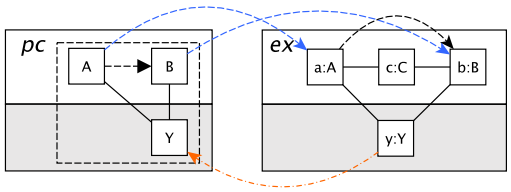


(b) Traceability links cannot be found

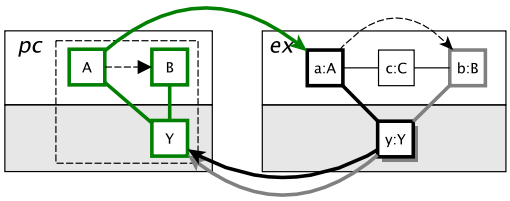
Fig. 20: Abstraction does not hold

In contrast to Figure 19, Figure 20 shows an example where the abstraction relation does not hold. Consider Figure 20a. 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.

As well, it is informative to examine Figure 20b. Note the one of the matches from the transformation execution attempts to match over 'a:A' and 'y:Y' elements, connected by a traceability link. Examination of the path condition shows that this traceability link is not present. Therefore, this path condition cannot accurately represent this transformation execution.



(a) Matching over match and apply graphs



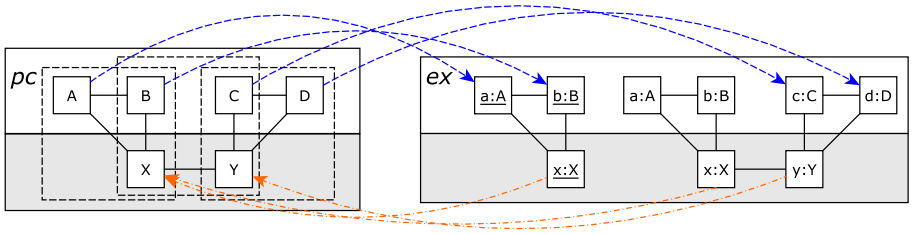
(b) Matching over traceability links

Fig. 21: Abstraction with indirect links

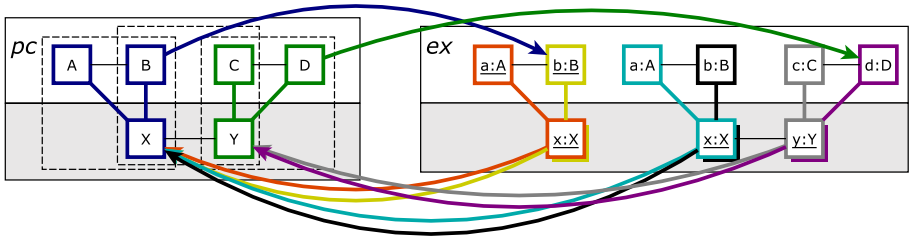
5.1.4 Example 4 – Indirect Links We now present a path condition in Figure 21a that includes indirect links. In this case, for the injective match to hold, the elements at both ends of the link must be found, and there must be an indirect link between the matched elements and between the elements in the transformation execution.

Note that the indirect link between elements $a : A$ and $b : B$ in the transformation execution is added by the containment transitive closure $Input^*$ in proposition 1 of Definition 28 to allow matching indirect links. Note also that, for the sake of our example, we are assuming that the links between $a : A$, $b : b$ and $c : C$ are containment relations.

Figure 21b highlights the structures involved in matching over traceability links. From the path condition, the structure contains the A, B, and Y elements with connected traceability links. From the transformation execution, there are two structures to be found in the path condition denoted in bold in the transformation execution. The matching of all structures can be successfully performed, and thus this abstraction relation holds.



(a) Abstraction holds on match and apply graphs



(b) Abstraction holds on traceability links

Fig. 22: Abstraction where path condition has combined rules

5.1.5 Example 5 – Combined Rules Figure 22 shows a path condition that is composed of a multitude of rules which have been combined in the path condition generation algorithm. Each individual rule is surrounded by dashed lines.

Note that the matching on the match and apply graphs in Figure 22a is similar to other examples. The combined rules can be considered as a single graph for the abstraction relation.

Figure 22b shows the matching when multiple traceability links are present in the transformation execution. Note that each individual traceability link and the connected elements are matched onto the path condition.

5.2 Validity and Completeness

In this section, we discuss the validity and completeness of the abstraction relation. Only proof sketches are presented here for Proposition 1 and Lemma 1, while more complete proofs are found in Appendix C, in the proofs of Proposition C.1 and Lemma C.1 respectively.

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

Proof sketch. Let $tr \in \textsc{Transf}^{sr}_{tg}$ be a DSLTans transformation. We wish to demonstrate that, for all path conditions $pc \in \textsc{PathCond}(tr)$, there exists a transformation execution $ex \in \textsc{Exec}(tr)$ of the set of rules used to build pc such that pc abstracts ex, as formally expressed in Definition 28. We can prove this property by induction on the set of transformations $\textsc{Transf}^{sr}_{tg}$ (see Definition 16), as follows:

- *Base case:* the base case is when tr is the empty transformation. In this case, according to Definition 27 only the empty path condition ε_{pc} exists in the path condition set. We thus need to demonstrate that the empty path condition abstracts the empty transformation execution ε_{ex}, as well as any execution for which the input model is never matched by any rule (consequently having an empty output model). For any of these transformation executions, Proposition 1 of the abstraction relation definition is satisfied, as no rule copy exists in the path condition and the output of the transformation execution is empty – empty typed graph homomorphisms thus satisfy the all the conditions of the proposition. Proposition 2 of the abstraction relation definition also trivially holds because no traceability links exist either in the path condition or in any of the considered executions.
- *Inductive case:* assuming every path condition generated for a transformation tr abstracts at least one transformation execution, we need to show that every path condition generated for a transformation tr', resulting from adding a layer $l \in \textsc{Layer}^{sr}_{tg}$ to tr, will also abstract at least one transformation execution.

In order to demonstrate the inductive case we need to show the property holds for all path conditions resulting from combining the rules of layer l with any path condition generated for tr. These path conditions for transformation tr' are built as expressed in Definition 25. According to this definition, path conditions for tr' are built by incrementally combining the path conditions generated for tr with a rule of layer l, until all the rules in l have been treated. We can thus again use induction for this proof, this time on the set of possible layers LAYER_{lg}^{sr}.

- *Base case:* this is the case where layer l contains no rules. In this case, by the base case of Definition 25, no new path condition is added to the set of path conditions generated for the transformation tr. As such the $tr = tr'$ and by induction hypothesis the property trivially holds for all path conditions generated for tr'.
- *Inductive case:* for the inductive case (transitive case of Definition 25) we need to show that, assuming the property holds for all path conditions generated for a transformation tr, then the property will also hold for a transformation tr' – where tr' results from adding a new rule rl to the last layer of tr. We will thus need to consider the four cases of rule combination:

  1. Rule rl has no dependencies (Definition 20).
  2. Rule rl has dependencies and cannot execute (Definition 21).
  3. Rule rl has dependencies and may and/or will execute (Definition 24).

  The property trivially holds for case 2, given that no new path conditions are added to the path condition set generated for tr and that the property holds for tr by induction hypothesis. When a rule rl is added to the last layer of tr such that cases 1 or 3 occur, then the property can be shown to hold for tr' as follows: 1) choose for a general path condition pc generated for tr an execution ex such that pc abstracts ex; 2) build an input model m as the result of uniting the input model of ex with a model that can be matched by rl; 3) execute tr' having as input model m to produce transformation execution ex'; and finally 4) demonstrate ex' is abstracted by the path condition pc' resulting from combining pc with rl whether rule rl does not depend on pc or rule rl depends on pc and may and/or will execute.

□

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

Proof. Let $tr \in \text{TRANSF}_{tg}^{sr}$ be a DSLTans transformation. We wish to demonstrate that, for all transformation executions $ex \in \text{EXEC}(tr)$, a path condition $pc \in \text{PATHCOND}(tr)$ exists such that ex is abstracted by pc, as formally expressed in Definition 28.

Completeness can be shown as a corollary of Proposition 1 about the *validity* of path condition generation. The complete set of executions $\text{EXEC}(tr)$ (see Definition 17) can be split into two kinds of executions:

1. The *empty execution* ε_{ex} or the *execution where the input model was not matched by any rule*. As mentioned in Proposition 1, these executions are abstracted by the empty path condition ε_{pc}.

2. The *execution ex where a number of rules of tr have been applied to the input model*, where each transformation rule rl of tr may have been applied more than once. In this case we have that, because all possible and valid rule combinations are considered when building path conditions, a path combination pc exists that contains one or more copies of each of the rules used when operationally building ex.

   Moreover, during the proof of *validity* of path condition generation in Proposition 1 we demonstrate that, when we add a new rule rl to the last layer of a transformation tr (such that we have a new transformation tr'), the rule combination step explained in Definitions 20, 21 and 24 produces a new set of path conditions where each path condition in that set still abstracts at least one transformation execution of tr'. This part of the proof (the second induction) is achieved by building for transformation tr' an input model m that can be matched by rl (as well as by all the other rules of tr), and then building from m a new transformation execution that is abstracted by a path condition built for tr'. Because in the proof of Proposition 1, m is such that it can be matched by rl an arbitrary amount of times, we know that, independently of the number of times a rule is applied during the construction of a transformation execution, a path condition abstracting that transformation execution exists.

   Additionally, input elements that are not matched by any rule do not affect the abstraction relation, as explained in case 1 above. This means we also know that executions

involving input models that are only partially matched by the rules of tr are also abstracted by one path condition.

□

Lemma 1 *(Uniqueness) A transformation execution is abstracted by exactly one path condition.*

Proof sketch. Let $tr \in \text{TRANSF}_{tr}^{sr}$ be a model transformation. We will demonstrate that two different path conditions $pc_1, pc_2 \in \text{PATHCOND}(tr)$ cannot exist such that we have a transformation execution $ex \in \text{EXEC}(tr)$ where $ex \Vdash pc_1$ and $ex \Vdash pc_2$.

We will do so by attempting to to build an $ex \in \text{EXEC}(tr)$ such that $ex \Vdash pc_1$ and $ex \Vdash pc_2$ and demonstrating that it is always the case that such is not possible. In order to structure our argumentation, we will consider two cases:

1. the case where no rules in tr have dependencies.
2. the case where some rules in tr have dependencies.

We start by considering that tr falls into case 1 above. By Definition 27 of path condition generation, each rule appears at most once in a path condition. Also, by construction, each path condition always contains a different combination of rules. We additionally know from Definition 16 that the rules that compose tr necessarily have non-overlapping matchers. We can nonetheless build a model m as the typed graph union of two input models m_1 and m_2, where injective typed graph morphisms can be found between the match parts of the rule copies that form pc_1, and m_1. Injective typed graph morphisms can be found as well between the match parts of the rules that form pc_2, and m_2. We thus know that injective typed graph morphisms can be found between the rule copies that compose pc_1 and pc_2, and m. This satisfies the first condition of Equation (1) in Definition 28 of abstraction relation.

Let us now consider that ex_1 and ex_2 are obtained by executing the transformations rules combined into pc_1 and pc_2, having m as input model. As mentioned above, we know that the rules in pc_1 and pc_2 are not completely overlapping. This means that, due to the way in which m is built (explained above), m will always have at least one input that is matched by rules of pc_1 but not by rules of pc_2 (and vice-versa). Thus, when the transformation rules combined into pc_1 execute having m as an input model, there will always exist a traceability link generated between an input and an output element of m that is not generated when the transformation rules combined into pc_2 execute having m as an input model (and vice-versa). As such, we have that ex_1 is always different from ex_2 by at least one traceability link. Given that this traceability link is symbolically represented in either pc_1 or pc_2 (but not in both), according to condition Equation (2) in Definition 28 it cannot be that either pc_1 or pc_2 abstract ex_1 and ex_2 simultaneously.

We will now analyse the scenario where tr falls into case 2 above, where some rules in tr have dependencies. For this case, assume we have a path condition pc_1 contained in the set of path conditions generated for tr, considering layers up to layer l of tr have executed. Assume also we have a rule rl of layer $l+1$ of tr that has dependencies and can be combined with pc. If rule rl is totally combined with path condition pc_1, according to Definition 24 and Figure 12b, then nothing needs to be shown as pc_1 is not kept in the path condition set but rather replaced by its combination with rl. However, in case rule rl is partially combined with pc, as defined in Definition 24 and Definition 22, then multiple path conditions are generated and additionally pc_1 is kept in the path condition set.

The proof will be complete once it is shown that for the path conditions that are generated when rules are partially combined, it is also the case that no two path conditions can abstract the same execution. This last part of the proof can be built in an analog fashion to the construction of the proof for point 1. As previously mentioned, the complete proof can be found as Lemma C.1 in Appendix C.

□

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 the transformation by examining the path conditions generated for it. We then rely on the abstraction relation presented in Definition 28 to extrapolate the proof result to all of the transformation's executions.

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: whenever this pattern is found in the input model, then this other pattern must be found in the output model, possibly including traceability constraints.

The property proving algorithm is relatively simple. The match part of a path condition includes a representation of all

the elements and relations "touched" in the input model of a set of transformation executions. Likewise, the property's pre-condition pattern represents the prerequisite for the property. Thus, the property proving algorithm will attempt to find the property's pre-condition pattern in the path condition's match graph. If not found, then the property will not be validated on this path condition, as the prerequisites do not exist.

Whenever the property's pre-condition pattern is found, the property's post-condition pattern is searched for in the path condition. If also found, then the rule execution(s) defined by that path condition will produce the required elements for that property and the property will hold. If not found, then the necessary elements will not be produced and the property check will fail for that path condition.

Path conditions for a transformation are checked to understand whether the property of interest holds on all of them. If it does, then by making use of the abstraction relation in Section 5 between path conditions and transformation executions, we can deduce that the property then holds for all transformation executions. If not, we deduce the property does not hold for at least one transformation execution. Later in this section we will develop a formal argument for why this is true.

6.1 Structure of a Property

We will now elaborate on the structures and the relations required for the property proving algorithm. Let us start by precisely defining what a property of a transformation is.

Definition 29 *Property of a Transformation*
Let $tr \in \text{TRANSF}_{tg}^{sr}$ *be a DSLTrans transformation. A property of* tr *is a 6-tuple* $\langle V, E, (s,t), \tau, Pre, Post \rangle$*, where* $Pre = \langle V', E', st', \tau' \rangle \in \text{IPATTERN}^{sr}$ *and* $Post = \langle V'', E'', st''\tau'' \rangle \in \text{IPATTERN}^{tg}$ *are indirect metamodel patterns. We also have that* $V = V' \cup V''$*,* $E \subseteq E' \cup E''$ *and* $\tau \subseteq \tau' \cup \tau''$*, where the co-domain of* τ *is the union of the co-domains of* τ' *and* τ'' *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* $\tau(e) = trace$*. Finally we have that there is at least one path condition* $\langle V_{pc}, E_{pc}, st_{pc}, \tau_{pc}, Match, Apply, Rule \rangle \in \text{PATHCOND}(tr)$ *for which a surjective typed graph homomorphism* $m \overset{f}{\blacktriangleleft} Pre$ *exists, where* $m \sqsubseteq Match$ *and* $f(v) \neq f(v')$ *if* v *and* v' *are elements of the path condition belonging to the same rule of set* $Rule$*. The set of all properties of transformation* tr *is called* $\text{PROPERTY}(tr)$.

In Definition 29, pre-conditions use the same pattern language as the match graph in DSLTrans rules, allowing the possibility of including several instances 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 pre-condition graphs.

Post-conditions also use the same pattern language as the apply graphs of DSLTrans transformation rules, with the additional possibility of expressing indirect links in post-conditions. Traceability links can also be used in properties to impose traceability relations between pre-condition and post-condition elements.

Note that Definition 29 includes a condition stating a surjective typed graph homomorphism needs to exist between the match part of at least one of the transformation's path condition, and the pre-condition of the property of interest. This condition makes sure that the property's pre-condition can be found at least in one execution of the transformation abstracted (the mathematical argument for this fact is given in the proof of Proposition 3). This condition makes the checking the validity of a property in the transformation meaningful. If this condition would not be true then it could be that the input pattern required by the property would never be fully matched during transformation execution, making such a property not relevant[3] for the transformation at hand.

6.2 Satisfaction of a Property

Let us now detail 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 found in the transformation execution.

Definition 30 *Satisfaction of a Property by an Execution of a Transformation*
Let $tr \in \text{TRANSF}_{tg}^{sr}$ *be a transformation. Let also* $p = \langle V, E, st, \tau, Pre, Post \rangle \in \text{PROPERTY}(tr)$ *be a property of* tr *and* $ex = \langle V', E', st', \tau', Input, Output \rangle \in Exec(tr)$ *be an execution of* tr*. Exe-*

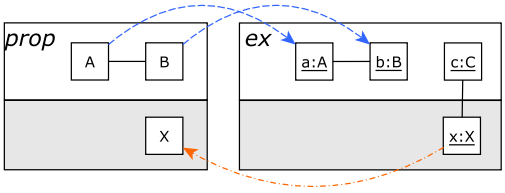
[3] In [6] we have referred to these properties *non-provable*. In the work presented here we explicitely disallow the construction of this class of properties.

cution ex satisfies property p, written ex $\models$ *p, if and only if:*

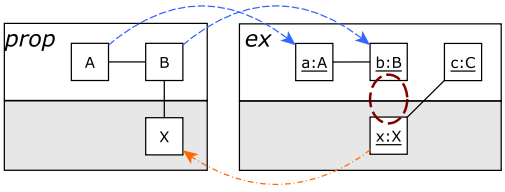
$$\forall f\ \exists g\,.\,\left(Pre \overset{f}{\triangleleft} Input^* \implies p \overset{g}{\triangleleft} ex^*\right)$$
$$where\ V(Input) \cap CoDom(g) = CoDom(f)$$

Definition 30 states that, every time a graph that is isomorphic to the property's pre-condition is found in (the containment transitive closure of) the input model of the transformation's execution, a graph that is isomorphic to the complete property needs to be found in (the containment transitive closure of) the transformation execution. Note that the last part of the proposition in Definition 30 ensures that the graph that is isomorphic to the property's pre-condition and the graph that is isomorphic to the complete property overlap on their pre-condition parts.

Figure 23a demonstrates how a property holds on a transformation execution. Note that the lack of traceability links in the property means no element creation dependencies have been specified. In contrast, the traceability links in the property in Figure 23b specify that the 'x:X' element must have been created from the 'b:B' element in the transformation execution. This is not the case (as highlighted by the dashed red circle), and therefore the property does not hold on the transformation execution.



(a) Property holds



(b) Property does not hold

Fig. 23: Matching property to a transformation execution

Due to the fact an infinite amount of transformation executions exists, proving the property directly on the set of transformation executions is not possible. We thus rely on the finite set of path conditions to prove properties about the set of all transformation execution. Let us then define what it means for a property to hold on, or be satisfied by, a path condition.

Definition 31 *Satisfaction of a Property by a Path Condition Let* $tr \in \text{TRANSF}^{sr}_{tg}$ *be a transformation. Let also* $p = \langle V, E, st, \tau, Pre, Post\rangle \in \text{PROPERTY}(tr)$ *be a property of* tr *and* $pc = \langle V', E', st', \tau', Match, Apply, Rulecop\rangle \in \text{PATHCOND}(tr)$ *be a path condition of* tr*. Path condition* pc *satisfies property* p*, written* $pc \vdash p$*, if and only if:*

$$\forall f\ \exists g\,.\,\left(in \overset{f}{\blacktriangleleft} Pre \implies out \overset{g}{\blacktriangleleft} p\right)$$
$$where\ in \sqsubseteq Match^* \ \wedge\ out \sqsubseteq pc^*$$

Additionally $Dom(g) \cap Match(pc^*) = Dom(f)$ *and* $f(v) \neq f(v')$, $g(v) \neq g(v')$ *whenever* v *and* v' *are elements of the path condition belonging to the same rule copy of set Rulecop.*

The principle behind the satisfaction relation in Definition 31 is the same as the one behind the satisfaction relation between a property and an execution of a transformation in Definition 30: whenever the property's pre-condition is found in the path condition then so is the complete property. Also, those two graphs found in the path condition share the property's pre-condition part. This last condition enforces that the pre- and post-conditions of the property are correctly linked by symbolic traceability links in the path condition.

Note that, despite their semantic similarity, the relations are expressed differently in Definition 30 and Definition 31. In Definition 30 – *satisfaction of a property by an execution of a transformation*, typed graph injective homomorphisms are defined from the property into the execution. However, in Definition 31 the direction of the typed graph surjective homomorphisms is from the path condition into the property. This can be explained by the fact, mentioned previously in this text, that different rules in a path condition may have elements that match over the same concrete instances of a transformation's input model. As such, we need to consider the case where match elements of a path condition, originating from different rules, overlap.

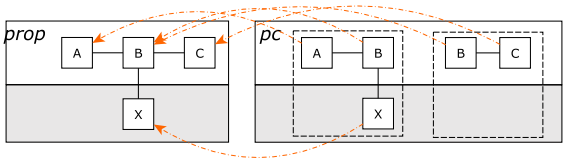


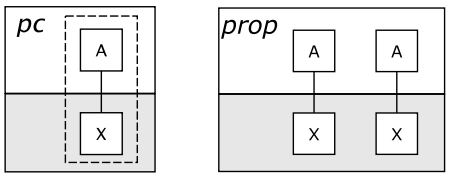
Fig. 24: Property satisfied by a path condition

This overlapping is modeled by the surjective typed graph homomorphisms of Definition 31 having the property as codomain. The surjections allow "forgetting" that two match elements of the path condition belong to different rules. Note however that these surjections are special, as two elements belonging to the same copy of a rule have to be mapped injectively onto the property. This situation is depicted in Figure 24. Note that element B is successfully matched even though it appears in two different rules in the path condition.

6.3 Expressiveness of the Property Language

As a result of taking rule combination (Section 4) and overlap (Section 6.2) into consideration, our technique allows proving properties of transformation executions that are matched and built by multiple rules. This is the main goal of our work, as the properties we are interested in regard all possible interactions of rules in a DSLTrans transformation. However, an expressiveness limitation of the property language exists: we cannot prove properties having pre-conditions that can be found by executing the exact same rule more than once. This is natural, as by definition our abstraction only considers one exemplar of each rule per path condition having the exact same type for each match element.

In order to illustrate this limitation, consider a transformation having one single rule that matches an elements of type A and that produces an element of type X. Our technique will create a path condition as seen in Figure 25a, abstracting over the number of times this rule has executed. According to the definition of satisfaction of a property by a path condition in Definition 31, the property in Figure 25b does not hold on the path condition in Figure 25a, although intuitively it should.



(a) Path condition created from a rule (b) Property with multiple instances of rule elements

Fig. 25: Example of unprovable property

Although this might be seen as a limitation of our technique, our case studies thus far indicate that very interesting properties exist that exclusively regard interactions between different rules. Nonetheless, our technique could be extended in order to consider more than one exemplar of the same rule per path condition. In fact, theoretically we already consider more than one exemplar of each rule when we treat polymorphism during path condition generation (see Definition 27). However, in this case all expanded rules for a rule *rl* are considered to be different as they differ by at least the type of one match element. Although this extension seems to fit relatively simply in our current theory, additional steps would need to be taken to understand which rules would be interesting to replicate to prove a particular property, and how many exemplars of each rule should be considered. In operational terms the number of replicas to consider of each rule is very important, given the exponential complexity of our path condition generation and property proof algorithms. Another possibility to tackle this issue would be to investigate and implement a more powerful satisfaction relation between path conditions and properties than the one we now present in Definition 31.

6.4 Validity and Completeness

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 if a property is satisfied by all path conditions generated for a transformation tr, then that property is satisfied by all executions of that transformation. On the other hand, if the property is not satisfied by at least one path condition, then it will not be satisfied by at least one transformation execution. In other words, we wish to show that no false positive or false negative proof results are induced by the abstraction relation.

On the other hand, *completeness* means that we are sure that all properties that can be expressed about a transformation can be shown to hold or not hold in all transformation executions.

As with the proofs for the validity and completeness of the abstraction relation, we present only proof sketches in this section in the interest of readability. Full proofs are shown in Appendix D as Proposition D.1 and Proposition D.2.

Proposition 3 *(Validity) The result of proving a property on a set of path conditions generated for a transformation or an all executions of that transformation is the same.*

Let $tr \in \text{TRANSF}^{sr}_{tg}$ *be a transformation and* $p \in \text{PROPERTY}(tr)$ *be a property of* tr*. Given this, we have that transformation*

tr satisfies property p if and only if:

$$\bigwedge_{pc \in \text{PATHCOND}(tr)} pc \vdash p \iff \bigwedge_{ex \in \text{EXEC}(tr)} ex \models p \quad (3)$$

Proof sketch. In order to prove the proposition in Equation (3) we will start by demonstrating that, if property p holds on a path condition pc generated for tr, then p will necessarily hold on all execution ex of tr that is abstracted by pc. On the other hand, if p does not hold on pc then it will not hold for at least one execution ex of tr abstracted by pc. This lemma can be stated as follows:

$$pc \vdash p \iff \forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} \,.\, ex \models p \quad (4)$$

We thus need to demonstrate both directions of the equivalence in Equation (4), as follows:

- The proof of the left-to-right direction of the equivalence is split into two cases as $pc \vdash p$ is true (as stated in Definition 31) when either: (1) the pre-condition of property p cannot be found in path condition pc; or (2) the pre-condition of p is found in pc and the post-condition of p also. We need to demonstrate that both these cases entail that $ex \models p$ holds.
- The proof of the right-to-left direction of the equivalence, which can be shown on its contrapositive:

$$\neg(pc \vdash p) \implies \exists ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} \,.\, \neg(ex \models p) \quad (5)$$

  According to Definition 31, to demonstrate Equation (5) we will need to show that when the pre-condition of property p is found in pc but the post-condition of p is not, the property does not hold on at least one execution of tr abstracted by pc.

Once Equation (4) is proved, we know that all path conditions on which a property holds represent executions on which the property also holds. Thus, if the property holds on all path conditions then it necessarily holds on all executions. On the other hand, if a property does not hold on one path condition, making it such that the conjunction on the left side of the equivalence in Equation (3) is false, then according to Equation (4) an execution for which it also does not hold exists. This makes it such that the conjunction on the right side of the equivalence in Equation (4) is also false. □

Proposition 4 *(Completeness) Properties of a transformation can be shown to either hold for all transformation executions, or not hold for at least one transformation execution.*

Proof. This results follows from two previous results: Proposition 2, that tells us that every transformation execution is abstracted by one path condition; and Proposition 3 that shows us that every path condition is taken into consideration during property proof. Note that Lemma 1 guarantees consistency of our results, in the sense that the uniqueness of one path condition per transformation execution guarantees that a property cannot be proven to be both *true* and *false* for two path conditions representing the same transformation execution. □

7 Implementation Details

In this section we will briefly describe our implementation of the algorithms described in the above sections. 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 [20,21] as our graph manipulation framework. Aside from satisfying our basic requirements described above, T-Core allows for native rewriting of typed graphs, which considerably eases our implementation effort. The algorithms described throughout this work have been implemented by scheduling T-Core graph manipulation primitives using the Python programming language.

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.

7.2.1 Path Condition Generation Recall that a DSLTrans transformation is composed of rules arranged in layers. The path condition generation algorithm described in Section 4 moves through these layers and combines rules into viable path conditions.

Let the number of rules in the transformation be r. Then, the maximum number of path conditions that can be created is 2^r. Each path condition will either represent a rule or not, and therefore the 2^r path conditions represent all possible rule combinations. Note that this case assumes that all path conditions are viable. In practice, unsatisfied rule dependencies will prevent some rule combinations, reducing the number of path conditions created.

As discussed in Section 4, the path condition generation algorithm builds these path conditions by considering all possibilities of how a rule can combine with a path condition. This combination step is composed of two algorithmic components. The first is to determine all positions where the rule matches over the path condition. Let this matching step be m. Note that this matching step is dependent on the size of the rules.

The second step of the combination step is to "glue" the rule at all matching positions. Let this step be termed g. This step is linear in the size of the rule to be glued multiplied by the number of times the rule has matched in step m.

Note that m and g could be quite expensive operations. However, in our implementation, these steps are implemented using the efficient T-Core graph manipulation framework.

$$O\big(2^r \cdot (m+g)\big) \tag{6}$$

Equation (6) presents the time and space complexity for the path condition generation algorithm.

7.2.2 Property Proof For our property proving algorithm, recall that each path condition created is examined to see if the property in question holds.

As mentioned in Section 6, a path condition must be matched by the property. However, different elements in the path condition may overlap (have been matched on) on the same elements in the input model, as described in Section 6.2. Therefore, an operational step is required to resolve this ambiguity. One solution is to produce all possible path conditions, where for each pair of overlapping elements in a path condition, one new path condition is produced where they are merged, and one new path condition where they are not.

The complexity of this "disambiguation step" will be proportional to the average number of overlapping elements in the path condition, and will be denoted by the term d in this discussion. Practically, d will be dependent on how rules are combined during the path condition generation algorithm. Future work will precisely detail how the characteristics of the transformation affect the algorithm's complexity.

The complexity of the matching step will then be linear in the size of the set of path conditions. The property matching step itself (p) will then be linear in the size of the property and path conditions. Again, in practice p is implemented using the T-Core framework.

$$O\big(2^r \cdot 2^d \cdot p\big) \tag{7}$$

Equation (7) shows time and space complexity for the property proving step.

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;
- In Section 7.2.2 we detail how the overlap of elements in a path condition can be operationally handled by producing two new path conditions for each pair of overlapping elements. Given this procedure is recursive and presents exponential time complexity, we have performed this "disambiguation" step only when strictly necessary: when performing property verification on a path condition that contains the elements in the property. Note that the fact 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;
- 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 also holds for B.

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 (6) and Equation (7) suggest 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 26.

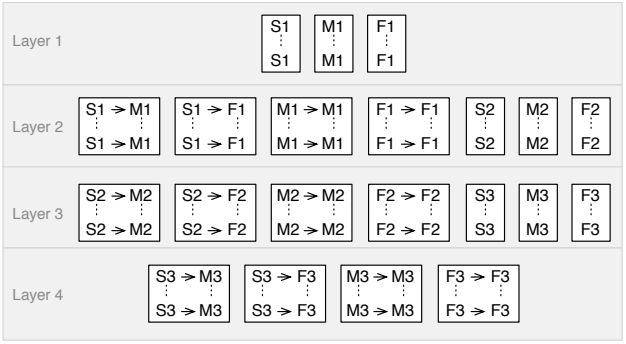


Fig. 26: Replicated Police Station transformation for performance tests

Note that for clarity reasons in Figure 26 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.

8.1.1 Results The results in Table 1 were obtained by verifying the properties in Figures 4a and 4b on the transformation seen in Figure 26. 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 [22].

8.1.2 Time Required to Produce Path Conditions An important metric for our work is measuring how long it takes

# 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

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 26. 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.

Figure 27a presents the number of path conditions created for a given number of rules, while Figure 27b 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 27c 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.

8.1.3 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 4 in Section 2. The first, in Figure 4a, is a property that we expect to hold for all path conditions. Figure 4b shows a property that we expect to *not* hold for all path conditions.

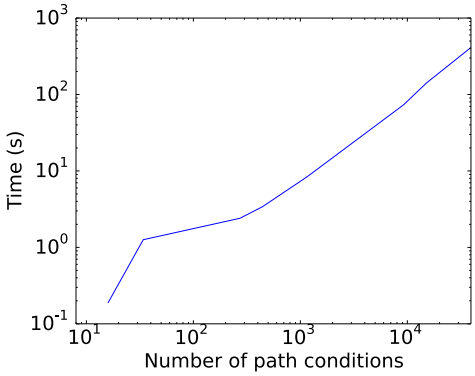
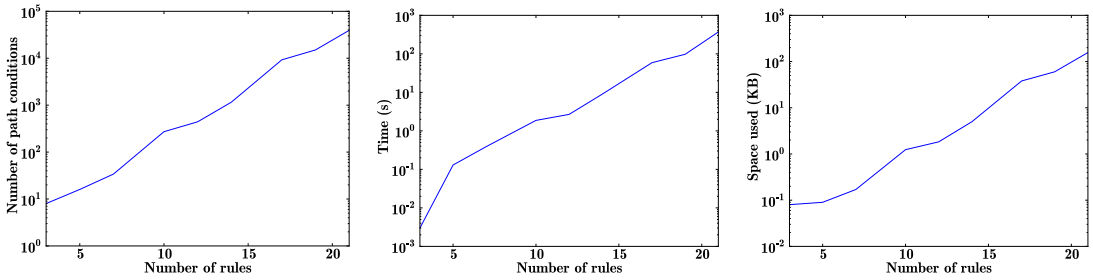


Fig. 28: Time required to prove the property that holds on all path conditions

Figure 28 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. The very short time to disprove the property is due to the fact that path conditions are checked for the property sequentially, in the order they are produced. In our example, a counterexample can be found very early in the set of path conditions. Note that a more complex property that involves rules which would appear only much later in the generated set of path conditions would require a longer time to reach a counter-example.



(a) Number of rules vs. path conds. created

(b) Number of rules vs. time taken

(c) Number of rules vs. memory used

Fig. 27: Metrics for the path condition creation process

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 metamodel from General Motors, describing legacy automotive configuration data, and the AUTOSAR metamodel, an open platform shared by car manufacturers. This DSLTrans mapping transformation includes seven rules, distributed among three layers. Further details of this experiment can be found in [5] and a complete description of the transformation can be found in [23].

Our path condition generation approach generates a set of three path conditions for the transformation in approximately 0.8 seconds. This low number of path conditions is due to the fact that several rules overlap, as explained in Definition 16 of Section 3. Such overlapping causes the number of formed path conditions to be smaller than in the case where no overlaps occur, as certain combinations need not be considered due to rule dependency.

In [18] we also describe the proof of nine properties (multiplicity invariants, security invariants and pattern contracts) that demonstrate several aspects of the correctness of this mapping transformation. The proof of these properties on all executions of a migration transformation is of interest to our industrial partners in order to ensure that the migration does not add extraneous elements or delete any needed information.

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. Note that, despite the fact that not all aspects of the case study (overlapping rules, propositional logic extension) are considered theoretically in this manuscript, the obtained results are nonetheless very interesting in terms of the experimental scalability of our approach. In particular, we note that our verification approach performs orders of magnitude faster compared to an ATL-based verification tool that verified the same transformation [18].

8.3 Discussion

The experimental results of verifying the test Police Station transformation portrayed in the graphs of Figure 27 show that, as predicted by complexity Equation (6), both the path condition construction time and the number of created path conditions grow exponentially with the number of rules. In Figure 28 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 fractions of 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. The complexity of Equation (6) 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 (6) 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 [24], AToM^3 [25], VIATRA2 [26], ATL [27] or VTMS [28]. As mentioned previously, DSLTrans' transformation are *terminating* and *confluent* by construction. This is achieved by expressiveness reduction which means that constructs which imply unbounded recursion or non-determinism are avoided. DSLTrans is, to the best of our knowledge, the only graph based transformation language where these properties are enforced by construction.

It is recognized in the literature that *termination* and *confluence* are important properties of model transformations, as these transformations have properties that are easier to understand and analyse. However, termination is undecidable for graph based transformation languages [29]. 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 [30–34]. Confluence is also undecidable for graph based transformation languages [35]. As for termination, several confluence criteria and corresponding analysis techniques have been proposed in the literature [36,34,37,38].

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 [28]. 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.* [39] 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 [40].

Also in 2012 Büttner *at 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 *at 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. First, the authors' proofs may require human assistance, depending on the used SAT solver. Also, despite the fact that Büttner *at al.* do treat constraints on object attributes, which we do not do, their results are presented for a small (6 rule) transformation and no scalability data, even preliminary, is presented. Finally, 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' ap-

proach 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, similarly 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 [41]. The authors formally prove these nested conditions have the expressiveness of first-order graph formulas. Poskitt and Plump later propose in [42] 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, which can specify, play, and analyse graph transformations [43]. 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 [44] 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 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 [45–47] 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 symbolically represent all transformation executions (resulting from the application of all rules in a DSLTrans transformation to any input model) which are in an abstraction relation with the path conditions, such that we are able to symbolically 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 [48]. 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 adapted symbolic execution techniques to verify DSLTrans model transformations. As well, 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.

As is the case in general for exhaustive verification methods, we have encountered theoretical and practical limitations when developing our technique. From a theoretical standpoint, not all DSLTrans transformations are currently addressed by the technique presented here. In particular, DSLTrans transformation where rules overlap in the match part (as per Definition 16 in Section 3) are not currently treated. Addressing overlapping rules theoretically implies some revisions to the formalisation presented here: on the one hand, rules that overlap imply rule dependency management during path condition construction; on the other hand, the uniqueness lemma in Lemma 1 of Section 5 needs to be re-analysed under looser constraints. Note however that we have already addressed this issue in practice in [5, 18] and that we expect the impact in the theory to be relatively small.

Another theoretical limitation has to do with the properties that can be proved using our technique. As expected, the chosen abstraction relation we use imposes limitations on which properties can be shown to hold or not hold on a DSLTrans transformation. In particular, because in general we only consider one rule copy in each path condition, we cannot prove properties where the pattern in the property implies the same rule matches on an input model more than once. We do not see this as a too strong limitation of our technique given that: on the one hand we are able to prove, for all executions of a DSLTrans transformation, a range of properties concerning the interaction between different rules in a DSLTrans transformation, which is where we expect most errors to occur; on the other hand we believe we can solve, at least partially, this property expressiveness problem and we have pointed some solutions to it in Section 6.3.

From a practical standpoint, we have shown with the two examples presented in Section 8 that there are good indicators that our technique can scale to transformations of practical interest. We have shown in Section 7 that the complexities of path condition generation and property proving are, as expected, exponential. Still, we are confident that we have not exhausted the set of possible optimizations in our tool and that our implementation (using typed graph manipulations in T-Core) can be made to scale well for reasonably-sized model transformations. This remains to be proved for larger model transformations. In this direction, we are currently implementing the analysis of a UML-RT to Kiltera transformation [49] which includes more than twice the number of rules in the industrial case study we present in this paper. For the UML-RT to Kiltera case study we are also including element attributes in the generation of path conditions and property proof.

Additionally, we have recently completed a propositional logic extension to our property language, which has already been used to express and prove meaningful properties in our industrial case study [5, 18]. This extension has been implemented in our tool, but its full impact in the theory of property proving, as explained in Section 6, is yet to be fully understood. 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, and their inclusion in the property language.

Acknowledgements

The authors would like to deeply thank Dániel Varró, Clark Verbrugge and the anonymous reviewers for their detailed and helpful comments. 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. Selim, G.M.K., Lúcio, L., Cordy, J.R., Dingel, J., Oakes, B.J.: Specification and Verification of Graph-Based Model Transformation Properties. In: ICGT, Springer (2014) 113–129 http://msdl.cs.mcgill.ca/people/levi/30_publications/files/paper_icgt_2014.pdf.
19. King, J.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7) (1976) 385–394
20. Syriani, E., Vangheluwe, H.: De-/Re-constructing Model Transformation Languages. ECEASST **29** (2010)
21. Syriani, E., Vangheluwe, H., LaShomb, B.: T-core: a framework for custom-built model transformation engines. Software and Systems Modeling (2013) 1–29
22. Lúcio, L.: SyVOLT: A Prototype Implementation (2013) http://msdl.cs.mcgill.ca/people/levi/files/SyVOLT.zip.
23. 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
24. Taentzer, G.: AGG: A Tool Environment for Algebraic Graph Transformation. In: AGTIVE. Volume 1779., Springer (2000) 333–341
25. De Lara, J., Vangheluwe, H.: AToM^3: A Tool for Multi-formalism and Meta-Modelling. In: FASE '02, Springer-Verlag (2002) 174–188
26. Varró, D., Pataricza, A.: Generic and Meta-transformations for Model Transformation Engineering. In: UML, Springer (2004) 290–304
27. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming **72**(12) (2008) 31 – 39
28. Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. Electronic Notes in Theoretical Computer Science **127**(1) (2005) 65–75
29. Plump, D.: Termination of Graph Rewriting is Undecidable. Fundamentae Informatica **33**(2) (1998) 201–209
30. De Lara, J., Vangheluwe, H.: Automating the Transformation-based Analysis of Visual Languages. Formal Aspects of Computing **22**(3-4) (May 2010) 297–326
31. 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)
32. 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
33. Bruggink, H.J.S.: Towards a Systematic Method for Proving Termination of Graph Transformation Systems. Electronic Notes in Theoretical Computer Science **213(1)** (2008)
34. Küster, J.M.: Definition and Validation of Model Transformations. SoSyM **5(3)** (2006) 233–259
35. Plump, D.: Confluence of Graph Transformation Revisited. In: Processes, Terms and Cycles: Steps on the Road to Infinity, Springer (2005)
36. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: ICGT, Springer (2002)
37. Lambers, L., Ehrig, H., Orejas, F.: Efficient Detection of Conflicts in Graph-based Model Transformation. Electronic Notes in Computer Science **152** (2006)
38. Biermann, E.: Local Confluence Analysis of Consistent EMF Transformations. ECEASST **38** (2011) 68–84
39. Guerra, E., Soeken, M.: Specification-driven Model Transformation Testing. Software & Systems Modeling (2013) 1–22

40. 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
41. 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
42. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundam. Inform. **118**(1-2) (2012) 135–175
43. 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
44. Rensink, A., Schmidt, A., Varró, D.: Model Checking Graph Transformations: A Comparison of Two Approaches. In: ICGT, Springer (2004) 226–241
45. Rensink, A., Distefano, D.: Abstract graph transformation. Electr. Notes Theor. Comput. Sci. **157**(1) (2006) 39–59
46. 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
47. Rensink, A., Zambon, E.: Pattern-based graph abstraction. Volume 7562 of Lecture Notes in Computer Science., Springer (2012) 66–80
48. 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
49. Posse, E.: Mapping UML-RT State Machines to Kiltera. Technical Report 2010-569, Queen's University, Kingston, Ontario, Canada (2010)

A Formal Background

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 A.1 *Typed Graph*
A typed graph is a 6-tuple $\langle V, E, (s,t), \tau, VT, ET \rangle$ *where:* V *is a finite set of vertices;* E *is a finite set of directed edges connecting the vertices* V*;* (s,t) *is a pair of functions* $s : E \to V$ *and* $t : E \to V$ *that respectively provide the source and target vertices for each edge in the graph; function* $\tau : V \cup E \to VT \cup ET$ *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 A.2 *Typed Graph Union*
Let $\langle V, E, (s,t), \tau, VT, ET \rangle, \langle V', E', (s',t'), \tau', VT', ET' \rangle \in \text{TG}$ *be typed graphs, where* VT *and* ET' *are disjoint sets, as well as* VT' *and* ET*. The typed graph union is the function* $\sqcup : \text{TG} \times \text{TG} \to \text{TG}$ *defined as:*

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

For the formal development of our technique, 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 A.3 *Typed Graph Homomorphism*
Let $\langle V, E, st, \tau, VT, ET \rangle = g$ *and* $\langle V', E', st', \tau', VT', ET' \rangle = g' \in \text{TG}$ *be typed graphs. A typed graph homomorphism between* g *and* g' *is a function* $f : V \to V'$ *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* $\tau(e) = \tau(e')$*. The domain of* f *is noted* $Dom(f)$ *and the co-domain of* f *is noted* $CoDom(f)$*. When an* injective *typed graph homomorphism* f *exists between* g *and* g' *we write* $g \overset{f}{\lhd} g'$*, or simply* $g \lhd g'$ *when the context is unambiguous. When a* surjective *typed graph homomorphism* f *exists between typed graphs* g *and* g' *we write* $g \overset{f}{\blacktriangleleft} g'$*, or also simply* $g \blacktriangleleft g'$ *in an unambiguous context.*

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 A.4 *Typed Subgraph*
Let $\langle V, E, st, \tau, VT, ET \rangle = g, \langle V', E', st, \tau', VT', ET' \rangle = g' \in \text{TG}$ *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 A.5 *Typed Graph Isomorphism*
Let $\langle V, E, st, \tau, VT, ET \rangle = g, \langle V', E', st', \tau', VT', ET' \rangle = g' \in \text{TG}$ *be typed graphs. We say that* g *and* g' *are isomorphic, written* $g \cong g'$*, if and only if there exists a bijective typed graph homomorphism* $f : V \to V'$ *such that* $f^{-1} : V' \to V$ *is a typed graph homomorphism.*

Notation: In order to simplify our notation, when the context is unambiguous we will abbreviate a typed graph $\langle V, E, st, \tau, VT, ET \rangle$ as as 4-tuple $\langle V, E, st, \tau \rangle$. Also, given a typed graph $g \in \text{TG}$, will use the notation $Components(g)$ to describe the set of strongly connected typed graphs in g. Finally, we will use the notation $g|_t$ to refer to the restriction of graph g to its subgraph containing only edges of type t.

B 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 B.1 *Metamodel*

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

A formal metamodel is a particular kind of typed graph where vertices represent classes and edges represent 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 A.1 in order to allow distinguishing between *abstract* and *concrete* classes, as well as between *containment* and *reference* edges in our metamodels. For simplification purposes, we do not model association cardinalities in our formal notion of metamodel as cardinalities are not strictly necessary in our development.

Definition B.2 *Expanded Metamodel*

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

- $V' = V \setminus \{v \in V \mid \tau(v) = (\cdot^4, abstract)\}$;
- $v_1 \xrightarrow{e} v_2 \in E'$ *if* $v_1 \xrightarrow{e} v_2 \in E$ *and* $\tau(v_1) = (\cdot, concrete)$ *and* $\tau(v_2) = (\cdot, concrete)$;
- *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 construct where all the relations between types of a metamodel are made explicit, rather than remaining implicit in the specialization hierarchy. It is built by adding to the original metamodel typed graph a relation of type t between two classes of the metamodel, whenever those classes specialize two classes that are also related by a relation of type t. Abstract classes and their relations do not carry over to the expanded metamodel. Expanded metamodels will be used in the subsequent text to facilitate formal the treatment of any structure involving polymorphism.

Definition B.3 *Metamodel Instance*

An instance of a metamodel $mm = \langle V',E',st',\tau',\leq\rangle \in$ META *is a typed graph* $\langle V,E,st,\tau\rangle \in$ TG*, where the co-domain of* τ *equals the co-domain of* τ'*. Also, there is a typed graph homomorphism* $f : V \to V'$ *from* $\langle V,E,st,\tau\rangle$ *to the expanded metamodel* $mm^\star$ *and the graph* $\langle V, \{e \in E \mid \tau(e) = (\cdot, containment)\}\rangle$ *is acyclic. The set of all instances for a metamodel mm is called* 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 cyclic containment relations, as enforced by EMF.

Definition B.4 *Containment Transitive Closure*

The containment transitive 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* $V = V'$*,* $\tau' \supseteq \tau$ *and* τ*'s co-domain is the union of the co-domain of* τ' *and the set* $\{indirect\}$*. We also have that* $E' = E \cup E^*_c$ *where* E^*_c *is the transitive closure of the set* $\left\{v \xrightarrow{e} v' \mid \tau(v \xrightarrow{e} v') = (\cdot, containment)\right\}$ *and if* $e \in E \setminus E'$ *then* $\tau(e) = indirect$*. We denote* 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 the definitions that follow we will use the $*$ notation, as in Definition B.4, to denote the containment transitive closure of structures that directly or indirectly include

[4] In our mathematical development we use a 'dot' notation to represent that we do not care about the value of a particular variable in a given context.

metamodel instances. For example, tg^* would represent the containment transitive closure of typed graph tg wherever containment edges are found in the graph. Note that the $*$ notation is different from the $\star$ notation, introduced in Definition B.2 for an expanded metamodel.

Definition B.5 *Model*

A model of a metamodel $mm = \langle V', E', st', \tau', \leq \rangle \in$ META *is a metamodel instance* $\langle V, E, st, \tau \rangle \in$ INSTANCE^{mm}*, such that: there exists an injective typed graph homomorphism* $f : V \to V'$ *from* $\langle V, E, st, \tau \rangle$ *to metamodel* $mm^\star$ *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* MODEL^{mm}.

A model, as per Definition B.5, 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. Note that this constraint does not necessarily lead to infinite models in the case of containment relations with the same source and target classes. In fact, if the cardinality of the target class is allowed to be zero, then it is not necessary that the containment relation is instantiated. This is for example the case of the containment relation *supervise* in the metamodel of Figure 1a.

Definition B.6 *Input-Output Model*

An input-output model is a 6-tuple $\langle V, E, (s,t), \tau, Input, Output \rangle$*, where:* $Input = \langle V', E', st', \tau' \rangle \in$ INSTANCE^{sr} *is a model;* $Output = \langle V'', E'', st'', \tau'' \rangle \in$ INSTANCE^{tg} *is a metamodel instance; Input and Output are disjoint. Additionally we have that* $V = V' \cup V''$*,* $E \subseteq E' \cup E''$ *and* $\tau \subseteq \tau' \cup \tau''$*, where the co-domain of* τ *is the union of the co-domains of* τ' *and* τ'' *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* IOM$^{sr}_{tg}$.

An input-output model is an object we will use when defining the semantics of a DSLTrans model transformation in Section 3.2. It is composed of two metamodel instances, one called the *input* and the other one the *output*. An input-output model allows the representation of intermediate operational 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.

Definition B.7 *Metamodel Pattern and Indirect Metamodel Pattern A pattern of a metamodel* $mm \in$ META *is an instance of* mm*. Given a metamodel pattern* $\langle V', E', st', \tau' \rangle \in$ INSTANCE^{mm} *we have that* $\langle V, E, st, \tau \rangle$ *is an* indirect *pattern if* $V = V'$*,* $E' \supseteq E$ *and the co-domain of* τ *is the union of the co-domains of* τ' *and the set* $\{indirect\}$*. Also, if* $e \in E \setminus E'$*, then we have that* $\tau(e) = indirect$*. Given a metamodel* mm*, the set of all metamodel patterns for* mm *is called* PATTERN^{mm}*. The set of all indirect metamodel patterns for* mm *is called* IPATTERN^{mm}.

Metamodel patterns are introduced in Definition B.7 as an intermediate notion, formally equal to metamodel instances. An *indirect* metamodel pattern is a metamodel pattern that includes edges typed as *indirect*. Both structures will be used as building blocks in the construction of transformation-related structures in the upcoming text.

Definition B.8 *Transformation Rule*

A transformation rule is a 6-tuple $\langle V, E, (s,t), \tau, Match, Apply \rangle$*, where:* $Match = \langle V', E', st', \tau' \rangle \in$ IPATTERN^{sr} *such that:* $Match \neq \varepsilon$[5] *is an non-empty indirect metamodel pattern;* $Apply = \langle V'', E'', st'', \tau'' \rangle \in$ PATTERN^{tg} *such that* $Apply \neq \varepsilon$ *is a metamodel pattern; Match and Apply are disjoint. We also have that* $V = V' \cup V''$*,* $E \subseteq E' \cup E''$ *and* $\tau \subseteq \tau' \cup \tau''$*, where the co-domain of* τ *is the union of the co-domains of* τ' *and* τ'' *and the set* $\{trace\}$*. 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* $\tau(e) = trace$*. 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* $\tau(e) = trace$*, or that* E'' *is not empty. The set of all transformation rules for a source metamodel* sr *and a target metamodel* tg *is called* RULE$^{sr}_{tg}$.

A transformation rule is the elemental block of a model transformation. Several transformation rules can be observed in the Police Station transformation in Figure 2. A formal 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 or an edge, meaning in practice that a rule will always produce

[5] We use the simplified ε notation to denote empty n-tuples structures.

something and not only match. A match pattern can include indirect links that are used to transitively match containment relations in 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 *trace*.

Definition B.9 *Matcher of 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 matcher, noted* $\|rl\|$, *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* $\tau(e) = trace$ *and* $V' = V_m \cup \{ v_1 \mid v_1 \xrightarrow{e} v_2 \in E \wedge \tau(e) = trace \}$.

Definition B.9 introduces the notion of matcher for a transformation rule which consists solely of the match pattern of a rule and its backward links, if any. The matcher of a rule constitutes the complete pattern that a DSLTrans rule attempts to match over a input-output model during rule execution. Traceability links between input and output model elements generated during transformation execution are matched by transformation rules' backward links, as informally explained in Section 2.2.

Definition B.10 *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* $rl^\star$ *is a set of transformation rules built as follows:*

- $rl \in rl^\star$;
- $\langle V, E, st, \tau', Match, Apply \rangle \in rl^\star$ *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 such that polymorphism is correctly handled in the developments that follow.

Definition B.11 *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* LAYER_{tg}^{sr}. *A model transformation is a finite list of layers denoted* $[l_1 :: l_2 :: \ldots :: l_n]$ *where* $l_k \in \text{LAYER}_{tg}^{sr}$ *and* $1 \leq k \leq n$, $n \in \mathbb{N}$. *We also impose that for any pair of rules* $rl_1, rl_2 \in \bigcup_{1 \leq k \leq n} l_k$, *if* $\|rl_1\| \cong rl$ *and* $rl \sqsubseteq \|rl_2\|$ *then* rl_2 *appears in a layer later than* rl_1 *and the apply parts of* rl_1 *and* rl_2 *are not isomorphic. The set of all transformations for a source metamodel s and a target metamodel t is called* TRANSF_{tr}^{sr}.

Definition B.11 formalises the abstract syntax of a model transformation, introduced at the beginning of this section. An example of a model transformation can be observed in Figure 2, the Police Station transformation. 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 B.11 imposes that, for any two pair of rules in the transformation, the matcher of the second rule never partially or totally subsumes (or contains) the matcher of the first rule, unless the second rule is in a subsequent layer and produces something more than the first rule. This condition avoids situations where the execution of a rule in a DSLTrans model transformation necessarily implies the execution of another rule (except for when rules having backward links necessarily execute because all their dependencies were created during the execution of previous layers).

B.0.1 ***Notation:*** We naturally extend to input-output models (Definition B.6), transformation rules (Definition B.8) and transformation executions (Definition B.16) the typed graph operations introduced in Section 3. Also, given a structure such as transformation rule $rl = \langle V, E, st, \tau, Match, Apply \rangle$, we will refer to the structure's components by using the component's name followed by the variable that holds the structure in between parenthesis. For example, we will write $V(rl)$ to designate the V component of rl or $Apply(rl)$ to designate rl's $Apply$ component. We also use a simplified notation to refer to the components of the input/output or match/apply typed graphs of input-output models and transformation rules. We refer to these structures' 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 τ. 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.

B.0.2 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 B.12 *Match Function*

Let $m_{in} \in \text{IOM}_{lr}^{sr}$ 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:*

$$match_{rl}(m_{in}) = \left\{ glue_{noind} \mid glue \sqsubseteq m_{in}^{*} \wedge glue \cong \|rl\| \right\}$$

where $glue \in \text{IOM}_{tr}^{sr}$ is an input-output model and $glue_{noind}$ is a version of glue where the indirect links have been removed.

The match function in Definition B.12 looks for subgraphs ($glue \sqsubseteq m_{in}^{*}$) of an input-output model that are isomorphic to the backward matcher of the given transformation rule ($glue \cong \|rl\|$). Note that the containment transitive closure of the input-output model (m_{in}^{*}) is considered such that indirect links in the rule can looked for in the input model. Additionally, indirect links need to be removed from the input-output models resulting from the match function ($glue_{noind}$). This is so 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 B.13. Its role is extend all model fragments found by matching a rule on a given input-output model, such that each of those fragments becomes 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 fragments found when matching the rule.

Definition B.13 *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 $apply$: $\text{IOM}_{tg}^{sr} \times \text{RULE}_{tg}^{sr} \rightarrow \text{IOM}_{tg}^{sr}$ *function is defined as follows:*

$$apply_{rl}(m_{in}) = \bigsqcup_{m_{glue} \in match_{rl}(m_{in})} trace_{a_\Delta}(m_{glue} \sqcup a_\Delta)$$

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_Δ is always disjoint from the m_{in} input-output model and also that any two instances of a_Δ used in the large union are always disjoint.

Partial function trace : $\text{IOM} \times \text{IOM} \nrightarrow \text{IOM}$ *is such that* $trace_{\langle V_\Delta, E_\Delta, st_\Delta, \tau_\Delta \rangle}(\langle V, E, st, \tau \rangle) = \langle V, E', st', \tau' \rangle$ *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 $v_1 \in Output(V_\Delta)$, $v_2 \in Input(V)$ and $\tau'(e) = trace$. Finally, rl_{noind} is a version of rl where indirect links have been removed.*

In Definition B.13 a_Δ 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 *trace* function builds traceability edges between newly created vertices of the output model in a_Δ 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_Δ other than the fact that its union with the sub-model m is isomorphic to rl_{noind}, the a_Δ 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 B.12 and B.13 are complementary: the former gathers all the fragments of an input-output model that are matched by a transformation rule; the latter glues on the output part of each of those fragments new objects and relations created by a transformation rule.

Definition B.14 *Layer Step Semantics*

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

$$\frac{}{\langle m_{in}, m_{glue}, \emptyset \rangle \xrightarrow{layerstep} m_{in} \sqcup m_{glue}}$$

$$\frac{rl \in l,\ apply_{rl}\ (m_{in}) = m_{rout}, \quad \langle m_{in}, m_{glue} \sqcup m_{rout}, l \backslash \{rl\} \rangle \xrightarrow{layerstep} m_{out}}{\langle m_{in}, m_{glue}, l \rangle \xrightarrow{layerstep} m_{out}}$$

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 B.14 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 built the result of executing layer l.

Definition B.14 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 B.14 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 B.15 *Transformation Step Semantics*

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

$$\frac{}{\langle m, [] \rangle \xrightarrow{trstep} m}$$

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

$$\text{where } l^\star = \bigcup_{rl \in l} rl^\star$$

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 B.15, the input-output model m_{inter} is the output of executing a given layer that is passed onto the next layer as input. Note that an empty input-output model (ε) is passed as the second argument to the *layerstep* relation in Definition B.15. This is because in Definition B.14 of layer step semantics, the second argument of the relation is used as an accumulator for the model fragments that are added to be added to the output of the previous layer once all the rules in the current layer have been executed. The transformation rules in a layer are expanded before execution ($l^\star$) such that polymorphism in the match elements can be handled (see Definition B.10).

Definition B.16 *Model Transformation Execution*

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

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

A model transformation execution is the input-output model $\langle V', E', st', \tau', input, output \rangle \in \text{IOM}^{sr}_{tg}$*, where* $output \in \text{INSTANCE}^{tg}$ *is an instance of the output metamodel. The set of all model transformation executions for transformation* tr *is written* $\text{EXEC}(tr)$*. A model transformation with an empty input model is noted* ε_{ex}*.*

Finally, as stated in Definition B.16, we consider a model transformation execution to be the input-output model (IOM) resulting from executing a set of rules on a starting IOM. This starting IOM includes the transformation's input in its input part and has an empty output part. The starting IOM represents the first step of the transformation when no rule has been executed yet. A transformation execution results from executing all the rules in a DSLTrans model transformation.

We now prove two important properties about executions of transformations expressed in the subset of DSLTrans presented in this paper: *confluence* and *termination*. The proofs are provided at a high level, given the fact that DSLTrans essentially enforces both these properties by construction of the semantics of DSLTrans.

Proposition 5 *Confluence*

Every model transformation execution is confluent up to typed graph isomorphism.

Proof. We want to prove that for every model transformation execution of a transformation $tr \in \text{TRANSF}^{sr}_{tg}$ having as input a model $input \in MODEL^s$, its output is always the same up to typed graph isomorphism.
If we assume an execution of the transformation is not confluent then this should happen because of non-determinism when the execution of a transformation is being built. Non-determinism happens during the construction of a transformation execution at two points:

1. in definition B.13, a_Δ is non-deterministic up to typed graph isomorphism, which does not contradict the proposition we are trying to prove.

2. in definition B.14 transformation rule *rl* is chosen non-deterministically from layer *l*. Thus, the order in which the transformation rules are treated is non-deterministic. However, by Definition B.14 rules in a layer execute independently, which means no-side effects from rule ordering influence the execution of rules in the same layer. Also, the increments to the transformation by each rule of a layer in Definition B.14 are united using $\sqcup$ (see Definition A.2), which is an operation that is commutative by construction and thus renders the transformation result from each layer deterministic.

Given there are no other sources of non-determinism when building the execution of a transformation, every model transformation execution is confluent up to typed graph isomorphism. □

Proposition 6 *Termination*
Every model transformation execution terminates.

Proof. Let us assume that there is a transformation execution which does not terminate. In order for this to happen there must exist a part in the construction of the execution of a transformation which induces an algorithm with an infinite amount of steps. We identify three moments when this can happen:

1. if definition B.14 of execution of a layer induces an infinite amount of steps. The only possibility for this to happen is if a layer has an infinite amount of transformation rules, which is a contradiction with definition B.11.
2. if definition B.15 of execution of a transformation induces an infinite amount of steps. Given layers are executed sequentially and no looping is allowed, the only possibility for this to happen is if the transformation has an infinite amount of layers, which contradicts definition B.11.
3. if the result of the $match_{rl}(m_{in})$ function in definition B.12 is an infinite set of match-apply graphs. The input-output model m_{in} is by definition finite and the matching of each rule is independent from the execution of other rules in the same layer. As such, the number of subgraphs of m_{in} isomorphic to *rl*'s matcher found during the execution of *rl* is finite.

Given there are no other constructs in the semantics of a transformation that can induce an infinite amount of steps, every model transformation execution terminates. □

C Validity and Completeness of Path Condition Generation

Definition C.1 *Path Condition*
A path condition is a 7-tuple $\langle V,E,(s,t),\tau,Match,Apply,Rulecop\rangle$, *where:* $Match = \langle V',E',st',\tau'\rangle \in \text{IPATTERN}^{sr}$ *is an indirect pattern;* $Apply = \langle V'',E'',st'',\tau''\rangle \in \text{PATTERN}^{tg}$ *is a pattern; Match and Apply are disjoint graphs. We also have that* $V = V' \cup V''$, $E \subseteq E' \cup E''$ *and* $\tau \subseteq \tau' \cup \tau''$ *where the co-domain of* τ *is the union of the co-domains of* τ' *and* τ'' *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 Rulecop component in the 7-tuple contains the set of rule copies used in the construction of the path condition, where each rule copy 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* $\text{PATHCOND}^{sr}_{tg}$ *and the empty path condition is noted* ε_{pc}.

Similarly to a transformation rule (see Definition B.8), 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 B.16). The path condition structure also contains a *Rulcop* set that allows identifying individually all the copies of rules that were used when building the path condition's typed graph. Note that we refer to *copies* of rules as, despite the fact that a path condition normally only contains one copy of each rule, in certain situations a rule may be used multiple times in the construction of a path condition. This will be explained further ahead in this section.

Notation: Given a path condition $pc = \langle V,E,st,\tau, Match,Apply,Rule\rangle \in \text{PATHCOND}^{sr}_{tg}$ we refer to the set of transformation rules in *pc* identified by the *Rule* relation as $Rule(pc)$. Also, because a path condition is particular kind of a typed graph, we naturally extend the basic notation of operators and homomorphisms on typed graphs defined in Section 3 to path conditions.

Definition C.2 *Combination of a Path Condition with a Rule*
Let $pc = \langle V',E',st',\tau',Match',Apply',Rulecop'\rangle \in \text{PATHCOND}^{sr}_{tg}$ *be a path condition and* $rl = \langle V'',E'',st'',\tau'', Match'',Apply''\rangle \in \text{RULE}^{sr}_{tg}$ *be a transformation rule, where their respective typed graphs can be joint. The union of pc with rl is built using the operator* $\overset{trace}{\sqcup} : \text{PATHCOND}^{sr}_{tg} \times \text{RULE}^{sr}_{tg} \rightarrow \text{PATHCOND}^{sr}_{tg}$, *as follows:*

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

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* $v_1 \in Apply(V'')$, $v_1 \notin Apply(V')$, $v_2 \in Match(V'')$ *and also that* $\tau'(e) = trace$. *Additionally,* $Match = Match' \sqcup Match''$ *and* $Apply = Apply' \sqcup Apply''$. *Finally, we have that:* $Rulecop = Rulecop' \cup rl$.

Definition C.2 shows the formal definition of combining a path condition with a rule. When a path condition is combined with a rule their typed graphs are united. Additionally, symbolic traceability links will be built at this time 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 links shared by the path condition and the rule graph. In the mathematical development that follows we will often refer to the joint parts of two or more typed graphs using the term "glue".

Definition C.3 *Path Condition and Rule Combination – No Dependencies*
The combination of a path condition pc and a rule rl, when rl has no dependencies, is described by the relation $\overset{combine}{\rightarrow} \subseteq \text{PATHCOND}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{RULE}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$, *defined as follows:*

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

Relation $\overset{combine}{\rightarrow}$ in Definition C.3 models the operational combination step. The relation has three input arguments: the first argument is the original path condition from the previous layer; the second argument is the set of path conditions accumulated thus far by combining other rules in the current layer with the original path condition; and the third argument is the rule from the current layer now being combined. The fourth argument of the relation, the relation's output, is the new set of path conditions resulting from this combination.Briefly, the equation in Definition C.3 states that whenever a rule has no backward links typed as *trace* (i.e. no dependencies), all path conditions in the accumulator set are kept, along with the result of combining all the path conditions in the accumulator set with the current rule.

Definition C.4 *Path Condition and Rule Combination – Unsatisfied Dependencies*
The combination of a path condition pc and a rule rl, when rl has dependencies that are not satisfied by pc, is described by the relation $\overset{combine}{\rightarrow} \subseteq \text{PATHCOND}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr}) \times \text{RULE}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr})$*, defined as follows:*

$$\frac{\neg\left(rl|_{trace} \blacktriangleleft pc|_{trace}\right)}{\langle pc, AC, rl\rangle \xrightarrow{combine} AC}$$

According to the pre-conditions of the equation presented in Definition C.4, a path condition does not satisfy the dependencies present in a rule if there is no surjective typed graph homomorphism between the backward links of the rule and the symbolic traceability links of the path condition. Besides expressing the fact that all backward links must exist as symbolic traceability links the path condition, the surjective homomorphism allows modeling the case where dependencies expressed by two (or more) backward links between similarly typed elements can be satisfied one single symbolic traceability link in the path condition.

Definition C.5 *Single Partial and Total Combination of a Set of Path Conditions with a Rule*
The single rule partial and total combination relations $\overset{p\_comb}{\rightarrow}$ *and* $\overset{t\_comb}{\rightarrow}$*, both having having signature* $\mathcal{P}(\text{PATHCOND}_{tg}^{sr}) \times \text{RULE}_{tg}^{sr} \times \text{RULE}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr})$ *are defined as follows:*

$$\frac{rl \cong rl_{glue} \sqcup ma_{\Delta}}{\langle AC, rl, rl_{glue}\rangle \xrightarrow{p\_comb} AC \ \cup \ \bigcup_{pc \in AC} pc \overset{trace}{\sqcup} (rl_{glue} \sqcup ma_{\Delta})} \quad \text{(C.1)}$$

$$\frac{rl \cong rl_{glue} \sqcup ma_{\Delta}}{\langle AC, rl, rl_{glue}\rangle \xrightarrow{t\_comb} \bigcup_{pc \in AC} pc \overset{trace}{\sqcup} (rl_{glue} \sqcup ma_{\Delta})} \quad \text{(C.2)}$$

Let us start by introducing relation $\overset{p\_comb}{\rightarrow}$, presented in Equation (C.1) of Definition C.5. The relation takes as arguments a set of path conditions being accumulated for the current layer, the rule to be combined, and an rl_{glue} argument indicating the place in each of the input path conditions the rule should be anchored to during the combination step. The relation's output is a new set of path conditions. This new set includes all the original path conditions, as well as each path condition in the accumulator set "glued" to a copy of rule being examined. Note that the relation $\overset{p\_total}{\rightarrow}$ in Equation (C.2) is similarly defined, except for the fact path conditions in the accumulator set are not preserved in the relation's output set.

Let us now define how a rule is combined with a path condition, whenever its backward links can be found several times in that path condition. We formalize it in Definition C.6, by means of relations $\overset{p\_step}{\rightarrow}$ and $\overset{t\_step}{\rightarrow}$. These two relations operationally describe the sequence of steps necessary to "glue" a rule at multiples places of a path condition. The set of places targeted in the path condition for receiving a copy of the rule is given by the sets *partialSet* and *totalSet* (found respectively in Equation (C.2) and Equation (C.4)). As expected, these sets contain the set of traceability links in the path condition where copies of the rule need to be anchored to.

Definition C.6 *Multiple Partial and Total Combination of a Set of Path Conditions with a Rule*
The multiple rule partial and total combination relations $\overset{p\_comb}{\rightarrow}$ *and* $\overset{t\_comb}{\rightarrow}$*, both having having signature* $\mathcal{P}(\text{PATHCOND}_{tg}^{sr}) \times \mathcal{P}(\text{RULE}_{tg}^{sr}) \times \text{RULE}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr})$ *are defined as follows:*

$$\frac{}{\langle AC, rl, \emptyset\rangle \xrightarrow{p\_step} AC} \quad \text{(C.1)}$$

$$\frac{rl_{glue} \in partialSet,\ \langle AC, rl, rl_{glue}\rangle \xrightarrow{p\_comb} AC'',\ \langle AC'', rl, partialSet \setminus \{rl_{glue}\}\rangle \xrightarrow{p\_step} AC'}{\langle AC, rl, partialSet\rangle \xrightarrow{p\_step} AC'} \quad \text{(C.2)}$$

$$\frac{}{\langle AC, rl, \emptyset\rangle \xrightarrow{t\_step} AC} \quad \text{(C.3)}$$

$$\frac{rl_{glue} \in totalSet\ ,\ \langle AC, rl, rl_{glue}\rangle \xrightarrow{t\_comb} AC'',\ \langle AC'', rl, totalSet \setminus \{rl_{glue}\}\rangle \xrightarrow{t\_step} AC'}{\langle AC, rl, totalSet\rangle \xrightarrow{t\_step} AC''} \quad \text{(C.4)}$$

Having Definition C.5 and Definition C.6 in mind, we can now proceed to define the complete combination relation of a rule with a path condition in the case of partially and totally satisfied dependencies.

Definition C.7 *Path Condition and Rule Combination – Partially and Totally Satisfied Dependencies*
The combination of a path condition pc and a rule rl, when rl has dependencies that are satisfied by pc, is described by the relation $\overset{combine}{\rightarrow} \subseteq \text{PATHCOND}_{tg}^{sr} \times \mathcal{P}(\text{PATHCOND}_{tg}^{sr}) \times$

$\text{RULE}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$, *defined as follows:*

$$\frac{\begin{array}{c} rl|_{trace} \blacktriangleleft pc|_{trace}\ , \\ \langle AC, rl, partialsat(rl,pc)\rangle \xrightarrow{p\_comb} AC''\ , \\ \langle AC'', rl, totalsat(rl,pc)\rangle \xrightarrow{t\_comb} AC' \end{array}}{\langle pc, AC, rl\rangle \xrightarrow{combine} AC'}$$

where

$$rl_{glue} \in partialsat(rl,pc) \iff rl_{glue} \sqsubseteq pc^* \wedge rl|_{trace} \blacktriangleleft rl_{glue} \ \wedge \ \nexists rl'.(rl_{glue} \sqsubseteq rl' \sqsubseteq pc^* \wedge \|rl\| \blacktriangleleft rl')$$

and

$$rl_{glue} \in totalsat(rl,pc) \iff rl_{glue} \sqsubseteq pc^* \wedge \|rl\| \blacktriangleleft rl_{glue}$$

The top equation in Definition C.7 defines the $\xrightarrow{combine}$ relation for when rule *rl* has dependencies that are satisfied by path condition *pc*. The pre-conditions in the equation state that the backward links in the rule are found in the path condition, as expected. Additionally, two sequential steps perform the gluing of the rule *rl* on all path conditions in accumulator *AC*, wherever the rule is partially and/or totally found in each of those path conditions. Relations $\xrightarrow{p\_comb}$ and $\xrightarrow{t\_comb}$ presented in Definition C.6 are used to model these two operational "gluing" steps. Functions *partialsat* and *totalsat*, described in the latter part of Definition C.7, are used to gather the places of path condition *pc* where copies of the rule need to be anchored to.

Definition C.8 *Combining a Path Condition with a Layer*
The layer combination relation $\xrightarrow{combpclayer} \subseteq \text{PATHCOND}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{LAYER}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$ *relation is defined as follows:*

$$\frac{}{\langle pc, AC, \emptyset\rangle \xrightarrow{combpclayer} AC}$$

$$\frac{\begin{array}{c} rl \in layer,\ \langle pc, AC\ , rl\rangle \xrightarrow{combine} AC'', \\ \langle pc, AC'', layer \setminus \{rl\}\rangle \xrightarrow{combpclayer} AC' \end{array}}{\langle pc, AC, layer\rangle \xrightarrow{combpclayer} AC''}$$

After the step in Definition C.8 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 process is modeled by relation $\xrightarrow{combpcsetlayer}$ in Definition C.9.

Definition C.9 *Combining a Set of Path Conditions with a Layer*
The path condition layer step relation $\xrightarrow{combpcsetlayer} \subseteq \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{LAYER}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$ *relation is defined as follows:*

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

$$\frac{\begin{array}{c} pc \in AC,\ \langle pc, \{pc\}\ , layer\rangle \xrightarrow{combpclayer} AC', \\ \langle AC \setminus \{pc\}, layer\rangle \xrightarrow{combpcsetlayer} AC'' \end{array}}{\langle AC, layer\rangle \xrightarrow{combpcsetlayer} AC' \cup AC''}$$

This working set of path conditions obtained for each layer 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 and is formally described in Definition C.10.

Definition C.10 *Path Condition Generation*
Let $[layer :: tr] \in \text{TRANSF}^{sr}_{tg}$ *be a transformation, where* $layer \in \text{LAYER}^{sr}_{tg}$ *is a Layer and tr also a transformation. The* $\xrightarrow{pathcondgen} \subseteq \mathcal{P}(\text{PATHCOND}^{sr}_{tg}) \times \text{TRANSF}^{sr}_{tg} \times \mathcal{P}(\text{PATHCOND}^{sr}_{tg})$ *is defined as follows:*

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

$$\frac{\langle \varepsilon_{pc}, \{\varepsilon_{pc}\}, layer^\star\rangle \xrightarrow{layercomb} AC''\ ,\ \langle AC'', tr\rangle \xrightarrow{pathcondgen} AC'}{\langle \varepsilon_{pc}, [layer :: tr]\rangle \xrightarrow{pathcondgen} AC}$$

$$\text{where } layer^\star = \bigcup_{rl \in l} rl^\star$$

Note that in Definition C.10, the recursive rule considers the expansion ($layer^\star$) of all the rules in a layer (see Definition B.10). This allows us to deal with polymorphism during path condition generation. In particular, given one rule *rl* of *layer*, we consider for path condition generation all rules containing possible of replacements of each match element in *rl* of certain type by an element belonging to one of the type's subtypes, as defined in the source metamodel *sr*.

Notation: We will use the abbreviation $\text{PATHCOND}(tr)$ to represent the set of path conditions *AC* produced for a transformation *tr*, where $\langle \varepsilon_{pc}, [layer :: tr]\rangle \xrightarrow{pathcondgen} AC$.

Definition C.11 *Abstraction of a Transformation Execution by a Path Condition*
Let $tr \in \text{TRANSF}^{sr}_{tg}$ *be a DSLTrans transformation. Let also*

$pc = \langle V, E, st, \tau, Match, Apply, Rules \rangle \in \text{PATHCOND}(tr)$ *of be a path condition of* tr *and* $ex = \langle V', E', st', \tau', Input, Output \rangle \in Exec(tr)$ *be an execution of* tr*. We have that* ex *is abstracted by* pc*, noted* $ex \Vdash pc$*, if and only if the set of transformation rules of* tr *combined in* pc *and the set of transformation rules of* tr *used to built* ex *is the same, and:*

$$\left(\forall rl \in Rules \,.\, Match(rl) \lhd Input^*\right) \wedge Output \blacktriangleleft Apply \tag{C.5}$$

and

$$\left(\forall trc \in Components(pc|_{trace}) \,.\, trc \lhd ex\right) \wedge \left(\forall e' \in E' \,\exists e \in E \,.\, \tau'(e') = trace \Longrightarrow \tau(e) = trace\right) \tag{C.6}$$

To understand the abstraction relation in Definition C.11, recall that during the construction of a transformation execution rules are matched injectively in the input model. This information is present in the first condition of the abstraction relation (Proposition C.5) via the injective typed graph homomorphism between the match part of the copies of rules "glued" onto the path condition and the containment transitive closure of the input part of the transformation execution. This relation enforces the fact that certain parts of the execution were found, or matched, by certain parts of the path condition. On the other hand, the surjection from the output of the execution towards the apply part of the path condition models the fact that the output of the execution has been completely built by instantiating the apply parts of the rules contained in the path condition.

The second condition of the abstraction relation (Proposition C.6) checks for the fact that symbolic traceability links in the path condition and traceability links in the execution correctly correspond to each other. This is modeled by the fact that all strongly connected components in the path condition, composed only of symbolic traceability links, are injectively found on the execution. This injection models the fact that traceability graphs between individual or combined rules in the path condition are necessarily found in the execution. Note that components of the path condition are considered because of the fact that disconnected rules in the path condition may have matched over common elements of a particular execution. As such, a full injection between the complete traceability graph in the path condition and the execution would be incorrect. Additionally, in the second part of Proposition C.6 we check the fact that every traceability link in the execution can be found in the path condition. This additional sanity check enforces that no spurious traceability links that could not have been created by the rules present in the path condition exist in the transformation execution.

Finally, the last clause of the abstraction relation states that rule copies that are repeated a number of times in the path condition need to be found at least a similar amount of times in the abstracted transformation execution.

It is important to mention that another abstraction relation, weaker or stronger, could have been chosen. The abstraction relation presented in Definition C.11 suits our needs in the sense that it allows us to demonstrate the validity and completeness of our proof technique, as we will show in the text follows. Additionally, it is particularly interesting because it makes sure that, given a DSLTrans transformation, each of its transformation executions is abstracted by one and only one of its path conditions. This result adds to the consistency of our theory and is also exposed later in this section.

Proposition C.1 *(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 pc abstracts ex (i.e. $ex \Vdash pc$), as formally expressed in Definition C.11. We can prove this property by induction on the set of transformations TRANSF_{tg}^{sr} (see Definition B.11), as follows:

- *Base case:* the base case is the case when we have $tr = [\,]$, i.e. the empty transformation. In this case, according to Definition C.10, only the empty path condition ε_{pc} exists in the path condition set. The empty path condition abstracts the empty transformation execution ε_{ex} (see Definition B.16), as well as any execution for which the input model is never matched by any rule (consequently having an empty output model). For any of these transformation executions, Equation (1) of the abstraction relation definition is satisfied, as: a) no rule copy exists in the path condition and the output of the transformation execution is empty – empty typed graph homomorphisms thus satisfy the all the conditions of the proposition; and b) Equation (2) of the abstraction relation definition trivially holds because no traceability links exist either in the path condition or in any of the considered executions.
- *Inductive case:* assuming every path condition generated for a transformation tr abstracts at least one transforma-

tion execution, we need to show that every path condition generated for a transformation tr', resulting from adding a layer $l \in \text{LAYER}_{tg}^{sr}$ to tr, will also abstract at least one transformation execution.

In order to demonstrate the inductive case we need to show the property holds for all path conditions resulting from combining the rules of layer l with any path condition generated for tr. These path conditions for transformation tr' are built as expressed in Definition C.8. According to this definition, path conditions for tr' are built by incrementally combining the path conditions generated for tr with a rule of layer l, until all the rules in l have been treated. We can thus again use induction for this proof, this time on the set of possible layers LAYER_{tg}^{sr} built as expressed in Definition B.11.

- *Base case:* this is the case where layer l contains no rules. In this case, by the base case of definition C.8, no new path condition is added to the set of path conditions generated for the transformation tr. As such the $tr = tr'$ and by induction hypothesis the property trivially holds for all path conditions generated for tr'.
- *Inductive case:* for the inductive case (transitive case of Definition C.8) we need to show that, assuming the property holds for all path conditions generated for a transformation tr, then the property will also hold for a transformation tr' – where tr' results from adding a new rule rl to the last layer of tr. We will thus need to consider the four cases of rule combination:

  1. Rule rl has no dependencies (Definition C.3).
  2. Rule rl has dependencies and cannot execute (Definition C.4).
  3. Rule rl has dependencies and may and/or will execute (Definition C.7).

  The property trivially holds for case 2, given that no new path conditions are added to the path condition set generated for tr and that the property holds for tr by induction hypothesis.

  When a rule rl is added to the last layer of tr such that cases 1 or 3 occur, new path conditions are added to the path condition set. Both cases are based on combining a path condition with a rule, as laid out in Definition C.2. In order to demonstrate this second inductive step we then need to show that, whenever the property holds for a path condition pc generated for a transformation tr, the combination of pc with a rule rl results in a new path condition where the property is respected.

  We start by picking for pc an execution ex such that pc abstracts ex. We know such a transformation execution exists by induction hypothesis. We can then build an input model m as the result of uniting the input model of ex with a model that can be matched by rl. If we execute tr' having m as input model we obtain transformation execution ex'.

  Let us now demonstrate ex' is abstracted by the path condition $pc' = pc \overset{trace}{\sqcup} rl$, the combination of pc with rl as shown in Definition C.2. We first recall the conditions of the abstraction relation in Definition C.11:

  1. a) injective typed graph homomorphisms must exist between the match parts of all the rule copies in the path condition and the input of the execution *and* b) a surjective typed graph isomorphism must exist between the output of the execution and the apply part of the path condition.
  2. a) injective typed graph homomorphisms must exist between all strongly connected components of the path condition composed of only symbolic traceability links *and* b) all isolated traceability links in the transformation execution must be found at least once in the path condition.

  Let us start by arguing for why condition 1 a) holds for pc' and ex'. Because we know rule rl has executed on the input model of ex', we know by Definition B.12 of the function matching a DSLTrans rule that an injective typed graph homomorphism exists between the match part of rl and ex'. When rl is combined with pc, its match part is preserved in pc and as such an injective typed graph homomorphism must exist between it and ex'. By induction hypothesis and because the combination operator is additive (meaning nothing existing in pc is deleted during combination) we know injective typed graph homomorphisms continue to exist between the match parts of all other rule copies in the path condition and the input of the ex'.

  In what concerns condition 1 b) above, we know by Definition B.13 and Definition B.14 that one or more copies of graphs isomorphic to the apply part of rl are added to the output of ex. Also, by Definition B.14, we know this addition preserves the output of ex and we also know by hypothesis that a surjective typed graph isomorphism exists between the output of ex and the apply part of pc. As

mentioned before, the combination of pc and rl is additive and as such we can also deduce that a typed graph isomorphism exists between the apply part of any copy rl added to ex and the apply part of rl added to pc. As such, all old and new edges and nodes in ex' can be surjectively found in pc'.

We will now discuss the reasons why condition 2 a) of the abstraction relation holds for pc' and ex'. When pc and rl are combined, by Definition C.2 a copy of the rule is "glued" on top of pc. Symbolic traceability links are added between elements of the match part of the copy of the rule and of the apply copy of the rule, for those elements in the apply part of the copy of the rule not previously connected to backward links. We also know by Definition B.13 that traceability links are similarly added to the copy of rl that is merged with ex. Because of the induction hypothesis we know that injective typed graph homomorphisms exists between all the strongly connected components composed of traceability links of pc and ex. When rl is combined with pc two cases can occur: a) rl has no backward links, in which the proposition trivially holds because isomorphic isolated strongly connected components are added both to pc and ex; b) rl has backward links, in which case the newly added components will connect to existing strongly connected components in pc and ex, forming additional strongly connected components. In this case, an injective typed graph homomorphism exists between each of the newly formed strongly connected components in pc and at least one newly formed strongly connected component in ex. This is so because, by Definition C.11 of the abstraction relation between a path condition and a transformation execution, the set of rules combined into pc and the set of rules that have executed is the same. The combination of these rules in the path condition, according to Definition C.2, replicates patterns that are produced in the execution by Definition B.13 and Definition B.14. Note that condition 2 a) of the abstraction relation provides additional guarantee that, when multiple partially and/or totally satisfied dependencies occur during path condition combination (Definition C.7), the corresponding executions are correctly abstracted given each place where the rules are "glued" corresponds to a different strongly connected component.

Condition 2 b) of the abstraction relation trivially holds as each new traceability link added to ex when rl executes has at least one corresponding symbolic traceability link in pc', resulting from the combination of pc with rl.

□

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

Proof. Let $tr \in \text{TRANSF}_{tg}^{sr}$ be a DSLTans transformation. We wish to demonstrate that, for all transformation executions $ex \in \text{EXEC}(tr)$, a path condition $pc \in \text{PATHCOND}(tr)$ exists such that ex is abstracted by pc, as formally expressed in Definition C.11.Completeness can be shown as a corollary of Proposition C.1 about the *validity* of path condition generation. The complete set of executions $\text{EXEC}(tr)$ (see Definition B.16) can be split into two kinds of executions:

1. The *empty execution* ε_{ex} or the *execution where the input model was not matched by any rule*. As mentioned in Proposition C.1, these executions are abstracted by the empty path condition ε_{pc}.

2. The *execution ex where a number of rules of tr have been applied to the input model*, where each transformation rule rl of tr may have been applied more than once. In this case we have that, because all possible and valid rule combinations are considered when building path conditions, a path combination pc exists that contains one or more copies of each of the rules used when operationally building ex.Moreover, during the proof of *validity* of path condition generation in Proposition C.1 we demonstrate that, when we add a new rule rl to the last layer of a transformation tr (such that we have a new transformation tr'), the rule combination step explained in Definitions C.3, C.4 and C.7 produces a new set of path conditions where each path condition in that set still abstracts at least one transformation execution of tr'. This part of the proof (the second induction) is achieved by building for transformation tr' an input model m that can be matched by rl (as well as by all the other rules of tr), and then building from m a new transformation execution that is abstracted by a path condition built for tr'. Because in the proof of Proposition C.1, m is such that it can be matched by rl an arbitrary amount of times, we know that, independently of the number of times a rule is applied during the construction of a transformation execution, a path condition abstracting that transformation

execution exists.

Additionally, input elements that are not matched by any rule do not affect the abstraction relation, as explained in case 1 above. This means we also know that executions involving input models that are only partially matched by the rules of tr are also abstracted by one path condition.

□

Lemma C.1 *(Uniqueness) A transformation execution is abstracted by exactly one path condition.*

Proof. Let $tr \in \text{TRANSF}_{tr}^{sr}$ be a model transformation. We will demonstrate that two different path conditions $pc_1, pc_2 \in \text{PATHCOND}(tr)$ cannot exist such that we have a transformation execution $ex \in \text{EXEC}(tr)$ where $ex \Vdash pc_1$ and $ex \Vdash pc_2$.

We will do so by attempting to to build an $ex \in \text{EXEC}(tr)$ such that $ex \Vdash pc_1$ and $ex \Vdash pc_2$ and demonstrating that it is always the case that such is not possible. In order to structure our argumentation, we will consider two cases:

1. the case where no rules in tr have dependencies.
2. the case where some rules in tr have dependencies.

We start by considering that tr falls into case 1 above. By Definition C.10 of path condition generation, each rule appears at most once in a path condition. Also, by construction, each path condition always contains a different combination of rules. We additionally know from Definition B.11 that the rules that compose tr necessarily have non-overlapping matchers. We can nonetheless build a model m as the typed graph union of two input models m_1 and m_2, where injective typed graph morphisms can be found between the match parts of the rule copies that form pc_1, and m_1. Injective typed graph morphisms can be found as well between the match parts of the rules that form pc_1, and m_2. We thus know that injective typed graph morphisms can be found between the rule copies that compose pc_1 and pc_2, and m. This satisfies the first condition of Equation (1) in Definition C.11 of abstraction relation.

Let us now consider that ex_1 and ex_2 are obtained by executing the transformations rules combined into pc_1 and pc_2, having m as input model. As mentioned above, we know that the rules in pc_1 and pc_2 are not completely overlapping. This means that, due to the way in which m is built (explained above), m will always have at least one input that is matched by rules of pc_1 but not by rules of pc_2 (and vice-versa). Thus, when the transformation rules combined into pc_1 execute having m as an input model, there will always exist a traceability link generated between an input and an output element of m that is not generated when the transformation rules combined into pc_2 execute having m as an input model (and vice-versa). As such, we have that ex_1 is always different from ex_2 by at least one traceability link. Given that this traceability link is symbolically represented in either pc_1 or pc_2 (but not in both), according to condition Equation (2) in Definition C.11 it cannot be that either pc_1 or pc_2 abstract ex_1 and ex_2 simultaneously.

We will now analyse the scenario where tr falls into case 2 above, where some rules in tr have dependencies. For this case, assume we have a path condition pc_1 contained in the set of path conditions generated for tr, considering layers up to layer l of tr have executed. Assume also we have a rule rl of layer $l+1$ of tr that has dependencies and can be combined with pc. If rule rl is totally combined with path condition pc_1, according to Definition C.7 and Figure 12b, then nothing needs to be shown as pc_1 is not kept in the path condition set but rather replaced by its combination with rl. However, in case rule rl is partially combined with pc, as defined in Definition C.7 and Definition C.5, then multiple path conditions are generated and additionally pc_1 is kept in the path condition set. Consider pc_2 is one of the newly created path conditions. In this case we can find a model m that can be injectively matched by the rule copies in both pc_1 and pc_2: m is the union of the input model isomorphic to the match part of pc_1, united with the input model isomorphic to the match part of pc_2 (including symbolic traceability links).

As before, we now consider that ex_1 and ex_2 are obtained by executing the rules used to build pc_1 and pc_2, respectively, having m as input model. In this case, we have that either rl was "glued" across different rule copies in pc_2, or over one single rule copy of pc_2. In the first case, by Definition B.8 of transformation rule we know either a new edge between output elements or a new output element have been produced in ex_2, but not in ex_1. According to the second part of Proposition 2 in Definition C.11 or the second part of Proposition 2 in Equation (1), this makes it such that it cannot be that either pc_1 or pc_2 abstract ex_1 and ex_2 simultaneously.

Finally, let us consider an additional path condition pc_3, also obtained from the partial combination of pc_1 with rl and where pc_3 is different of pc_1. In this case we have that pc_2 and pc_3 resulted from the combination of exactly the same rules, with the difference that certain rules have been "glued"

at more locations of one path condition than of the other. We can thus build a model m that can be injectively matched by the rule copies in both pc_1 and pc_2: the model is isomorphic to the the match part of the path condition (including symbolic traceability links) that has been "glued" more copies of rl upon. When we now obtain ex_2 and ex_3 by executing the rules in pc_2 and pc_3, we will have that one of these executions will necessarily contain more copies of rl's apply pattern than the other. Given the fact that these copies will necessarily have been "glued" over different strongly connected graphs of pc_2 and pc_3 (because rules having no dependencies do not overlap as explained for case 1), there cannot be an injective typed graph homomorphisms between all the strongly connected components formed by the traceability graphs of at least one of path conditions pc_2 or pc_3, and ex_1 (likewise for ex_2). Given this is required by the first part of Proposition 2 in Equation (1) of the abstraction relation, it cannot be that either pc_1 or pc_2 abstract ex_1 and ex_2 simultaneously.

□

D Validity and Completeness of Property Verification

Definition D.1 *Property of a Transformation*
Let $tr \in \text{TRANSF}^{sr}_{tg}$ be a DSLTrans transformation. A property of tr is a 6-tuple $\langle V, E, (s,t), \tau, Pre, Post\rangle$, where $Pre = \langle V', E', st', \tau'\rangle \in \text{IPATTERN}^{sr}$ and $Post = \langle V'', E'', st''\tau''\rangle \in \text{IPATTERN}^{tg}$ are indirect metamodel patterns. We also have that $V = V' \cup V''$, $E \subseteq E' \cup E''$ and $\tau \subseteq \tau' \cup \tau''$, where the co-domain of τ is the union of the co-domains of τ' and τ'' 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 $\tau(e) = trace$. Finally we have that there is at least one path condition $\langle V_{pc}, E_{pc}, st_{pc}, \tau_{pc}, Match, Apply, Rule\rangle \in \text{PATHCOND}(tr)$ for which a surjective typed graph homomorphism $m \overset{f}{\blacktriangleleft} Pre$ exists, where $m \sqsubseteq Match$ and $f(v) \neq f(v')$ if v and v' are elements of the path condition belonging to the same rule of set Rule. The set of all properties of transformation tr is called* $\text{PROPERTY}(tr)$.

In Definition D.1, pre-conditions use the same pattern language as the match graph in DSLTrans rules, allowing the possibility of including several instances 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 pre-condition graphs.

Post-conditions also use the same pattern language as the apply graphs of DSLTrans transformation rules, with the additional possibility of expressing indirect links in post-conditions. Traceability links can also be used in properties to impose traceability relations between pre-condition and post-condition elements.

Note that Definition D.1 includes a condition stating a surjective typed graph homomorphism needs to exist between the match part of at least one of the transformation's path condition, and the pre-condition of the property of interest. This condition makes sure that the property's pre-condition can be found at least in one execution of the transformation abstracted (the mathematical argument for this fact is given in the proof of Proposition 3). This condition makes the checking the validity of a property in the transformation meaningful. If this condition would not be true then it could be that the input pattern required by the property would never be fully matched during transformation execution, making such a property not relevant[6] for the transformation at hand.

Definition D.2 *Satisfaction of a Property by an Execution of a Transformation*
Let $tr \in \text{TRANSF}^{sr}_{tg}$ be a transformation. Let also $p = \langle V, E, st, \tau, Pre, Post\rangle \in \text{PROPERTY}(tr)$ be a property of tr and $ex = \langle V', E', st', \tau', Input, Output\rangle \in Exec(tr)$ be an execution of tr. Execution ex satisfies property p, written $ex \models p$, if and only if:

$$\forall f\ \exists g\,.\,\left(Pre \overset{f}{\lhd} Input^* \implies p \overset{g}{\lhd} ex^*\right) \quad \text{where } V(Input) \cap CoDom(g) = CoDom(f)$$

Definition D.2 states that, every time a graph that is isomorphic to the property's pre-condition is found in (the containment transitive closure of) the input model of the transformation's execution, a graph that is isomorphic to the complete property needs to be found in (the containment transitive closure of) the transformation execution. Note that the last part of the proposition in Definition D.2 ensures that the graph that is isomorphic to the property's pre-condition and the graph that is isomorphic to the complete property overlap on their pre-condition parts.

Definition D.3 *Satisfaction of a Property by a Path Condition*
Let $tr \in \text{TRANSF}^{sr}_{tg}$ be a transformation. Let also $p = \langle V, E, st, \tau, Pre, Post\rangle \in \text{PROPERTY}(tr)$ be a property of tr and $pc = \langle V', E', st', \tau', Match, Apply, Rulecop\rangle \in \text{PATHCOND}(tr)$ be a path condition of tr. Path condition pc satisfies property p, written $pc \vdash p$, if and only if:

$$\forall f\ \exists g\,.\,\left(in \overset{f}{\blacktriangleleft} Pre \implies out \overset{g}{\blacktriangleleft} p\right) \quad \text{where } in \sqsubseteq Match^* \wedge out \sqsubseteq pc^*$$

Additionally $Dom(g) \cap Match(pc^*) = Dom(f)$ and $f(v) \neq f(v')$, $g(v) \neq g(v')$ whenever v and v' are elements of the path condition belonging to the same rule copy of set Rulecop.

The principle behind the satisfaction relation in Definition D.3 is the same as the one behind the satisfaction relation between a property and an execution of a transformation in Definition D.2: whenever the property's pre-condition is found in the path condition then so is the complete property. Also, those two graphs found in the path condition share

[6] In [6] we have referred to these properties *non-provable*. In the work presented here we explicitely disallow the construction of this class of properties.

the property's pre-condition part. This last condition enforces that the pre- and post-conditions of the property are correctly linked by traceability link in the path condition.

Proposition D.1 *(Validity) The result of proving a property on a set of path conditions generated for a transformation or an all executions of that transformation is the same.*
Let $tr \in \text{TRANSF}^{sr}_{tg}$ *be a transformation and* $p \in \text{PROPERTY}(tr)$ *be a property of* tr*. This given, we have that transformation* tr *satisfies property* p *if and only if:*

$$\bigwedge_{pc \in \text{PATHCOND}(tr)} pc \vdash p \iff \bigwedge_{ex \in \text{EXEC}(tr)} ex \models p \quad \text{(D.1)}$$

Proof. In order to prove the proposition in Equation (D.1) we will start by demonstrating that, if property p holds on a path condition pc generated for tr, then p will necessarily hold on any execution ex of tr that is abstracted by pc. On the other hand, if p does not hold on pc then it will not hold for at least of one execution ex of tr abstracted by pc. This lemma can be stated as follows:

$$pc \vdash p \iff \forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} \,.\, ex \models p \quad \text{(D.2)}$$

We thus need to demonstrate both directions of the equivalence in Equation (D.2). On the one hand we need to prove of the left-to-right direction of the equivalence:

$$pc \vdash p \implies \forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} \,.\, ex \models p \quad \text{(D.3)}$$

Proposition D.3 is shown to be true in Lemma D.1. We then need to show the right-to-left direction of the equivalence:

$$\forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} \,.\, ex \models p \implies pc \vdash p \quad \text{(D.4)}$$

Proposition D.4 is shown to be true in Lemma D.2. Once propositions D.3 and D.4 are proved, we know that all path conditions on which a property holds represent executions on which the property also holds. Thus, if the property holds on all path conditions then it necessarily holds on all executions. On the other hand, if a property does not hold on one path condition, making it such that the conjunction on the left side of the equivalence in Equation (D.1) is false, then according to Equation (D.2) an execution for which it also does not hold exists. This makes it such that the conjunction on the right side of the equivalence in Equation (D.2) is also false. □

Lemma D.1 *If a property holds for a path condition then the property holds for any transformation execution that path condition abstracts.*
Let tr *be a transformation,* $pc \in \text{PATHCOND}(tr)$ *be a path condition of* tr*,* $ex \in \text{EXEC}(tr)$ *be an execution of* tr *and* $p \in \text{PROP}(tr)$ *be a property of* tr*. Then we have that:*

$$pc \vdash p \implies \forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} \,.\, ex \models p \quad \text{(D.5)}$$

Proof. By Definition D.3 we know that $pc \vdash p$ is equivalent to proposition $\forall f \; \exists g \,.\, \left(in \overset{f}{\blacktriangleleft} Pre \implies out \overset{g}{\blacktriangleleft} p\right)$, where Pre is p's pre-condition, in is a subgraph of the containment transitive closure of the match part of pc, and out is a subgraph of the containment transitive closure of pc. Additionally, by Definition D.2 we also know that $ex \models p$ is equivalent $\forall f \; \exists g \,.\, \left(Pre \overset{f}{\triangleleft} Input^* \implies p \overset{g}{\triangleleft} ex^*\right)$, where $Input$ is the input part of ex.

We will show that the implication holds by analysing the three cases where, $pc \vdash p$, the left side of Proposition D.5 holds.

1. If the precondition of the property cannot be found in the match part of a path condition pc, then it cannot be found in the input part of an execution abstracted by pc. Formally, we have that, assuming ex is abstracted by pc:

$$\nexists f \,.\, \left(in \overset{f}{\blacktriangleleft} Pre\right) \implies \nexists f' \,.\, Pre \overset{f'}{\triangleleft} Input^* \quad \text{(D.6)}$$

   where, as before, $Input^*$ is the containment transitive closure of of the input part of ex and in is a subgraph of the match part of pc. Proposition D.6 holds because of the fact that the surjection between in and Pre is defined such that it is in fact a set of injective typed graph homomorphisms between subgraphs of in belonging to different rule copies that compose pc, and Pre. We know such a set of injective typed graph homomorphisms cannot be found from in into Pre. However, the abstraction relation in Definition C.11 states that an injective typed graph homomorphism exists between each rule copy in the match part of pc and $Input^*$. We thus know that there cannot exist an injective typed graph homomorphism between Pre and $Input^*$.
2. For certain executions, the property holds on the path condition but the property's pre-condition cannot be found in

the execution.

$$\forall f\ \exists g\,.\,\left(in \overset{f}{\blacktriangleleft} Pre \wedge out \overset{g}{\blacktriangleleft} p\right) \implies \nexists f'\,.\,\left(Pre \overset{f'}{\lhd} Input^*\right) \tag{D.7}$$

These are the executions where a set of injective typed graph homomorphisms can be found from *in* into *Pre*, but not from *in* into $Input^*$, as required by the abstraction relation. If this is the case then this means that at least two vertices of *in* belonging to different rule copies that were mapped by f into the same vertex of *Pre*, are mapped into different vertices of $Input^*$ by f' (or vice-versa).

3. For the remaining set executions abstracted by *pc*, if the property holds on the path condition then the property holds on the execution. Formally, according to Definition D.3 and Definition D.2 we have that:

$$\begin{aligned}\forall f\ \exists g\,.\,\left(in \overset{f}{\blacktriangleleft} Pre \wedge out \overset{g}{\blacktriangleleft} p\right) \implies \\ \forall f'\ \exists g'\,.\,\left(Pre \overset{f'}{\lhd} Input^* \wedge p \overset{g'}{\lhd} ex^*\right) \\ \text{where } Dom(g) \cap Match(pc^*) = Dom(f) \text{ and} \\ V(Input) \cap CoDom(g') = CoDom(f')\end{aligned} \tag{D.8}$$

This is the case where every two vertices of *in* belonging to different rule copies that were mapped by f into a common vertex of *Pre*, are also mapped into a common vertex of $Input^*$ by f'. We thus need to show that the fact that the post-condition of the property holds on the path condition implies that the post-condition of the property also holds on the execution, i.e. that $out \overset{g}{\blacktriangleleft} p \implies p \overset{g'}{\lhd} ex^*$. This proposition is true because we know by Definition C.11 of abstraction relation that a surjective typed graph homomorphism exists between the output part of *ex* and the apply part of *pc*. By composing this surjection with the surjection between *out* and *p* we take as hypothesis, we know a surjective typed graph homomorphism exists between the output of *ex* and *p*. The inverse of this composed homomorphism contains an injective typed graph homomorphism between *p*'s post-condition and *ex*. We are thus missing accounting for the traceability links between the pre- and post-condition of property *p*, if they exist. According to Proposition D.8 we know that *in* and *out* overlap on their subgraphs that are isomorphic to *p*'s pre-condition. By Definition C.11 of the abstraction relation, we know that an injective typed graph homomorphism can be found between each strongly connected component formed of symbolic traceability links of *pc*, and *ex*. We also know that a typed graph surjective homomorphism exists between *out* and *p*. We thus know that the traceability links between the pre- and post-condition of *p* can be injectively found in *ex*. Note that strongly disjoint connected symbolic traceability link components mapped from *pc* to *ex* may be mapped onto joined traceability link components in *ex* when disjoint vertices of the match part of *pc* are mapped onto the same input vertex in *ex*.

The three cases above cover all executions that can be abstracted by a path condition, and as such we know that if the property holds on a path condition, it will necessarily hold on all the executions that path condition abstracts. □

Lemma D.2 *If a property holds for a transformation execution then the property holds for the path condition that abstracts it.*
Let tr be a transformation, $pc \in \text{PATHCOND}(tr)$ be a path condition of tr, $ex \in \text{EXEC}(tr)$ be an execution of tr and $p \in \text{PROP}(tr)$ be a property of tr. Then we have that:

$$\forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\}\,.\,ex \models p \implies pc \vdash p \tag{D.9}$$

Proof. We will demonstrate Proposition D.9 holds by contraposing it:

$$\neg(pc \vdash p) \implies \exists ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\}\,.\,\neg(ex \models p) \tag{D.10}$$

By Definition D.3 we know that $pc \vdash p$ is equivalent to proposition $\forall f\ \exists g\,.\,\left(in \overset{f}{\blacktriangleleft} Pre \implies out \overset{g}{\blacktriangleleft} p\right)$, where *Pre* is *p*'s pre-condition, *in* is a subgraph of the containment transitive closure of the match part of *pc*, and *out* is a subgraph of the containment transitive closure of *pc*. We also know by Definition D.2 that $ex \models p$ is equivalent $\forall f\ \exists g\,.\,\left(Pre \overset{f}{\lhd} Input^* \implies p \overset{g}{\lhd} ex^*\right)$, where *Input* is the input part of *ex*. After replacing the left and the right hand side of Proposition D.10 by equivalent formulas and solving the negations we reach the conclusion we need to prove:

$$\begin{aligned}\exists f\ \forall g\,.\,\left(in \overset{f}{\blacktriangleleft} Pre \wedge \neg(out \overset{g}{\blacktriangleleft} p)\right) \implies \\ \exists ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\}\,. \\ \exists f'\ \forall g'\,.\,\left(Pre \overset{f'}{\lhd} Input^* \implies \neg(p \overset{g'}{\lhd} ex^*)\right)\end{aligned} \tag{D.11}$$

We thus need to demonstrate that whenever the pre-condition of the property is found at least once in a path condition, but not its corresponding post-condition, then the same thing hap-

pens for at least one of the executions abstracted by that path condition. We know by Proposition D.11 that $in \overset{f}{\blacktriangleleft} Pre$, i.e. the precondition of the property is found at least once in the path condition. We thus know that there exists one execution for which $Pre \overset{f'}{\lhd} Input^*$ holds, which is the execution for which the surjective typed graph homomorphism f maps vertices belonging to the match parts of different rule copies in the same fashion that the set of injective typed graph homomorphisms from the abstraction relation in Definition C.11 maps to the match part of pc onto $input^*$.

In order to complete the proof we need to show that the fact that $\neg(out \overset{g}{\blacktriangleleft} p)$, i.e. if the complete property cannot be found in the path condition, then $\neg(p \overset{g'}{\lhd} ex^*)$, i.e. the complete property cannot be found in the execution. Note that, according to Definition D.3 and Definition D.2, we know the considered complete property graphs both in p and ex found by g and g' are anchored on the pre-condition graphs of the property found by f and f'. Because of the abstraction relation, we know a surjective typed graph homomorphism between the output of ex^* and the apply part of pc exists. Given a surjective typed graph homomorphism does not exist between pc and p, we know certain vertices and/or edges that exist in p, either in its apply part or in its symbolic traceability links, do not exist in pc. If the missing vertices and/or edges are part of the $apply$ part of p then we are sure an injective typed graph homomorphism cannot exist between p and ex because ex also does not contain those vertices or edges. If the missing edges are symbolic traceability edges then, according to the condition on strongly connected components in the abstraction relation in Definition C.11, we know that the traceability links in ex can be surjectively mapped onto pc. Because some of those traceability links are missing in p, an injective typed graph homomorphism cannot exist between p and ex.

□

Proposition D.2 *(Completeness) Properties of a transformation can be shown to either hold for all transformation executions, or not hold for at least one transformation execution.*

Proof This results follows from two previous results: Proposition C.2, that tells us that every transformation execution is abstracted by one path condition; and Proposition D.1 that shows us that every path condition is taken into consideration during property proof. Note that Lemma C.1 guarantees consistency of our results, in the sense that the uniqueness of one path condition per transformation execution guarantees that a property cannot be proven to be both *true* and *false* for two path conditions representing the same transformation execution.