

Symbolic Verification of Translation Model Transformations

Levi LÚCIO¹, Bentley James OAKES¹, and Hans VANGHELUWE^{2,1}

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.

Reviewer ► On Page 2, left, mid-page you state that your methods differ from previous work in that you require no intermediate representation. It is certainly not preordained that this difference is an advantage: such an intermediate representation typically gives access to general-purpose tooling, e.g., for model checking or theorem proving. You would have to argue that you are better off reimplementing that functionality in your own setting. Is there any evidence for that? An example of such evidence (in a somewhat different context) is for instance provided by Penemann in his PhD thesis, where by experiments he shows that his dedicated, graph-based resolution outperforms that of a general-purpose theorem prover. ◀

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 opportuni-

ties 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: ?? briefly introduces the DSLTrans model transformation language and its formal semantics, while the formal background for this work is presented in Section 2. The algorithms to build the complete set of path conditions for the transformation will be discussed in Section 5. Section 4 will present the abstraction relation found in our technique, along with examples, while Section 7 will examine how this abstraction relation is used in our process for proving properties. In Section 8 and Section 9, we introduce our implementation with sample scalability results; Section 10 presents the related work; and finally in Section 11 we conclude with remarks and future work.

Reviewer ► The paper is inconsistent (maybe due to the iteration of submitted versions) in what it says about its own structure. Page 2, right, 1st paragraph states that the language and its semantics are presented in Section 2; Section 2 (1st paragraph) says that the semantics is presented in Appendix B; most of it is actually in Section 3. Appendix B indeed repeats essentially all of Section 3, a duplication which certainly serves no purpose and should be remedied. In fact, I would much prefer all of Appendix B to be moved to the main text, as the only part that is now only in the appendix, the definition of a transformation, is absolutely central to the paper. As it currently stands, Section 3.4 has a similar problem: first it states that the semantics is not in the main text, then it refers to Section 3.2 for the semantics (which the reader has just gone through), then in Def. 17 jumps to a notation that is only introduced in Appendix B. ◀

2 Formal Background

In this section we will introduce the formal concepts that will be used throughout all this paper. We start in ?? 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.

Reviewer ► *Your formalisation is made more complex than necessary because you do not put the power of typed graphs to work. If your source and target metamodels have disjoint types (which you can assume without loss of generality) you can take their union, augmented with trace edge types from all target node types to all source node types, and take that to be the meta-model for an input-output model, and also for a rule. Suddenly you dont have to carry around the distinction between Match and Apply (for rules) or input and output (for input-output models) around any more: they are just the projections onto the types of the source, resp. target metamodel.* ◀

Reviewer ► *Section 4.2.2 on backward links is rather confusing, as it speaks about traceability links being added to rules. You never add traceability links to rules anywhere in the paper; in fact, your formal notion of a rule has no room for them. Instead, rules only have backward links. (The clarity of the the situation is not improved by the fact that those are labelled trace.)* ◀

Reviewer ► *I think the situation would be much more clear if you were to use the common notion of left hand side and right hand side of a rule. The left hand side of a rule rl is precisely your rule matcher, the right hand side is the entire rl augmented by your traceability links, as in 4.2.2. Suddenly we are back in a well-known graph transformation formalism, and your rules are in fact so simple (no deletion, no negative application conditions, injective matching) that I believe it makes no difference whether you use algebraic graph transformation or some more constructively defined variant. The only standard notion is that of transitive closure in the left hand side; but there are plenty of GT tools that offer this extension. In fact I see no reason why your rule application would then not precisely coincide with the the same notion in, say, SPO graph rewriting; and I hope you agree that this*

would help you no end in explaining what you are doing. (If, on the other hand, there is after all some difference then this, too, would be interesting to know, as that difference is not at all apparent right now.) ◀

Reviewer ► *Viewed like this, moreover, I think your notions of input-output models and rules are very close to triple graph transformation as used by Schrr at all. I think the main (if not only) difference lies in the fact that your "glue graph" is actually not a graph; instead, you use traceability edges directly from the target model to the source model. If you were to turn those edges into nodes with source and target edges, even that difference would disappear.* ◀

Armed with the fundamental notion of typed graph, we can then introduce other formal concepts in Sections 3, ?? and 3.7 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.

Reviewer ► *There is a lot of redundancy in the paper, on several levels: - input-output-models, path conditions and properties are all defined very similarly. Couldn't you subsume them in a single definition and spell out the differences?* ◀

Bentley ► *Make sure that terminology is consistent between elements and vertices* ◀

2.1 Typed Graphs

Typed Graph 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 $s : E \rightarrow V$ and $t : E \rightarrow V$ that respectively provide the source and target vertices for each edge in the graph
- Function $\tau : V \cup E \rightarrow VT \cup ET$ is a typing function for the elements of V and E , where VT and ET are disjoint finite sets of vertex and edge type identifiers and $\tau(v) \in VT$ if $v \in V$ and $\tau(e) \in ET$ if $e \in E$
- Edges $e \in E$ are noted $v \xrightarrow{e} v'$ if $s(e) = v$ and $t(e) = v'$, or simply e if the context is unambiguous
- The set of all typed graphs is called TG
- We define the empty graph to be a graph with all elements to be empty functions or sets

2.1.1 Vertex and Edge Types Note that the our verification technique is geared toward model-driven engineering. Therefore, the types of vertices and edges in our typed graphs will be drawn from a particular *metamodel*. Sample metamodels for our running example can be seen in figures above **Bentley** ► **Fix** ◀. Note that DSLTrans rules (to be formally defined) will combine typed graphs with potentially different metamodels.

We assume that this metamodel provides the necessary vertex and edge types, with any collisions resolved by adding to the metamodel name. As well, we assume the presence of a partial ordering \leq on the vertex and edge types for subtyping information.

For convenience, we define some utility functions in our formalization to aid the treatment of typing:

- $isAbstract : VT \rightarrow \{true, false\}$
 - Where $isAbstract(VT)$ returns *true* iff VT is denoted as abstract (not able to be instantiated) by the metamodel, else *false*
- $isIndirect : ET \rightarrow \{true, false\}$
 - Where $isIndirect(ET)$ returns *true* iff ET is denoted as an *indirect edge*, else *false*. The *indirect* classification allows matching over indirect paths between vertices. This will be further clarified when required for our constructs.
- $matchesOver : \{VT \cup ET\} \times \{VT \cup ET\} \rightarrow \{true, false\}$

- The purpose of this function is to assist in handling polymorphism in the metamodel, and to handle matching for the special DSLTrans types
- $matchesOver(T, T) \rightarrow true$
- $matchesOver(T, T') \rightarrow true$ iff T' is a subclass of T in some defined partial ordering \leq given by the metamodel
- Otherwise, *false*

This typing information is our implementation of a typed graph conforming to a metamodel. Note that for simplification purposes, we will not represent edge cardinalities or containment relationships given by a metamodel in our notion of typed graph. In fact we require these conditions to be relaxed to perform our graph rewriting.

Typed Subgraph 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. This will be needed to partition DSLTrans constructs such as rules into logical components.

Definition 2 *Typed Subgraph*

Let $\langle V, E, (s, t), \tau, VT, ET \rangle = g$, and

$\langle V', E', (s', t'), \tau', VT', ET' \rangle = g'$, where $g, g' \in TG$.

g' is a typed subgraph of g , written $g' \sqsubseteq g$, iff $V' \subseteq V$, $E' \subseteq E$ and $\tau' = \tau|_{V' \cup E'}$.

We also define a function $getEdges : ET \rightarrow \text{Set}(\text{Edges})$ to return all edges of a certain edge type.

Bentley ► **Where is this used?** ◀

Typed Graph Union 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 3 *Typed Graph Union*

The typed graph union is the function $\sqcup : TG \times TG \rightarrow TG$ defined as:

$$\begin{aligned} \langle V, E, (s, t), \tau, VT, ET \rangle \sqcup \langle V', E', (s', t'), \tau', VT', ET' \rangle = \\ \langle V \cup V', E \cup E', (s \cup s', t \cup t'), \tau \cup \tau', VT \cup VT', ET \cup ET' \rangle \end{aligned}$$

Note: as a reviewer helpfully pointed out, we require that $s \cup s'$, $t \cup t'$, and $\tau \cup \tau'$ coincide on common elements. However, this can be assumed w. l. o. g..

Bentley ► *Do we need typed graph intersection? Or to mention what it means for two graphs to be disjoint?* ◀

Bentley ► *Is this used in the DPO approach?* ◀

2.2 Homomorphisms

For the formal development of our technique, we are interested in relations between typed graphs that preserve some graph structure and vertex/edge type, i.e. homomorphisms.

However, our technique of symbolic execution of transformation rules means that we cannot rely on a trivial homomorphism. Instead our matching must be flexible about how vertices in the pattern graph may match over multiple vertices in the target graph.

Typed Graph Homomorphism The first typed graph homomorphism we define is standard, where the structure of the pattern graph must be found in the target graph. This is presented for the reader to appreciate the complications required for the next homomorphisms.

Definition 4 Typed Graph Homomorphism

Let $\langle V, E, (s, t), \tau, VT, ET \rangle = g$, and

$\langle V', E', (s', t'), \tau', VT', ET' \rangle = g'$, where $g, g' \in \text{TG}$.

A typed graph homomorphism from g onto g' is a function $f : f_v \cup f_e$ such that:

- $f_v : V \rightarrow V'$
- $f_e : E \rightarrow E'$
- $\forall v_1 \xrightarrow{e} v_2 \in E$:
 - Let $v'_1 = f_v(v_1), v'_2 = f_v(v_2), e' = f_e(e)$
 - $v'_1, v'_2 \in V', e' \in E'$
 - $s'(e') = v'_1, t'(e') = v'_2$
 - $\text{matchesOver}(\tau(v_1), \tau(v'_1)) \wedge$
 $\text{matchesOver}(\tau(v_2), \tau(v'_2)) \wedge$
 $\text{matchesOver}(\tau(e), \tau(e'))$

When an *injective*¹ typed graph homomorphism f exists from g onto g' , we write $g \xrightarrow{\text{inj}} g'$. When a *surjective* typed graph homomorphism f exists from g onto g' we write $g \xrightarrow{\text{surj}} g'$.

¹ For a terminology refresh, *injective matching* means there is a one-to-one correspondence between the pattern and target vertices. *Surjective matching* means that all target vertices must be matched by at least one pattern vertex. A *bijection* is when the matching is both injective and surjective.

Typed Graph Edge Homomorphism Our technique also requires a form of graph homomorphism that primarily focuses on matching the edges of the graphs. This homomorphism will be an injective match regarding the edges, but note that a particular vertex in the pattern graph may match onto multiple vertices in the target graph. **Bentley** ► *What's the term for this?* ◀

Definition 5 Typed Graph Edge Homomorphism

Let $\langle V, E, (s, t), \tau, VT, ET \rangle = g$, and

$\langle V', E', (s', t'), \tau', VT', ET' \rangle = g'$, where $g, g' \in \text{TG}$.

A typed graph edge homomorphism between g and g' is a function $h : h_v \cup h_e$ such that:

- $h_v : V' \rightarrow V$ Note that this function has unusual properties:
 - A function from a vertex in the target graph onto a vertex in the pattern graph
 - A partial function, as not all vertices in the target graph will be reflected in the pattern graph
 - Surjective, so that all pattern vertices are matched to by target vertices
 - A minimal function, such that the minimum number of target vertices are matched to each pattern vertex

Bentley ► *Not sure if this is needed.* ◀

- $h_e : E \rightarrow E'$, and is injective
- $\forall v_1 \xrightarrow{e} v_2 \in E$:
 - Let $e' = h_e(e), v'_1 = s'(e'), v'_2 = t'(e')$
 - $v'_1, v'_2 \in V', e' \in E'$
 - $v_1 = h_v(v'_1), v_2 = h_v(v'_2)$
 - $\text{matchesOver}(\tau(v_1), \tau(v'_1)) \wedge$
 $\text{matchesOver}(\tau(v_2), \tau(v'_2)) \wedge$
 $\text{matchesOver}(\tau(e), \tau(e'))$

Again, we wish to highlight the fact that this homomorphism is different than Definition 4 in the vertex matching function. A vertex in the target graph matches onto a vertex in the pattern graph, with multiple target vertices matching onto the same pattern vertex. **Bentley** ► *Create diagram to illustrate* ◀

This unusual homomorphism allows our technique to ‘split’ our pattern graph over multiple locations in the target graph. This is critical, as our target graphs are constructed in a way that allows for vertex duplication. That is, two vertices of a particular type in the target graph may only represent one underlying element, as discussed in Section 4. Constructing this morphism allows our prover implementation to avoid explic-

itly creating ‘disambiguated’ graphs where these duplications are resolved.

When an *injective* typed graph edge homomorphism f exists from g onto g' , we write $g \xrightarrow{inj} g'$. When a *surjective* typed graph edge homomorphism f exists from g onto g' we write $g \xrightarrow{surj} g'$.

Typed Graph Isomorphism

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 6 Typed Graph Isomorphism

Let $\langle V, E, (s, t), \tau, VT, ET \rangle = g$, and $\langle V', E', (s', t'), \tau', VT', ET' \rangle = g'$, where $g, g' \in \text{TG}$.

We say that g and g' are isomorphic, written $g \cong g'$, iff there exists a bijective typed graph homomorphism $f : V \rightarrow V'$ such that $f^{-1} : V' \rightarrow V$ is a typed graph homomorphism.

Transitive Closure DSLTrans matching constructs allow for the matching of indirect links, where as described in Section 3.2, rules may match over indirect paths between elements. This transitive closure allow for the explicit creation of all edges implied by these indirect links to aid in matching.

Definition 7 Transitive Closure

Let g be a graph $\langle V, E, (s, t), \tau \rangle \in \text{TG}$. Then the transitive closure $g^* = \langle V, E', (s', t'), \tau', VT, ET' \rangle \in \text{TG}$, where:

- $E' = E \cup v_1 \xrightarrow{e''} v_2 \mid \exists v_1 \xrightarrow{e_i} v_i, v_i \xrightarrow{e_j} v_j, \dots, v_k \xrightarrow{e_k} v_2 \wedge \text{isIndirect}(e_i)$
- **Bentley** ► **How to create s, t , and τ ?** ◀

Given a graph, its transitive closure includes, besides the original graph, all the edges belonging to the transitive closure of indirect links in that graph. Note that these transitive edges are identified by the function *isIndirect*.

In the definitions that follow we will use the $*$ notation, as in Definition 7, to denote the transitive closure of our structures.

2.2.1 Link Homomorphism

- Backward links injectively match from contract into PC onto traceability links. Source and target of backward links on contract must match the source and target of the traceability link of the PC.

We must also define a function to match link structures, such as backward links and the traceability links (to be discussed).

Definition 8 Link Homomorphism

Consider two sets of links $L = \{E_L, (s_L, t_L)\}$, and $T = \{E_T, (s_T, t_T)\}$. Note that these could be backward links or traceability links.

We define an injective homomorphism function f to match L over T , such that:

- $\forall e_L \in E_L :$
- $\forall e_T \in E_T \mid f(e_L) = e(T)$

Note that the typing of these links will be given by the component in which they are found.

2.2.2 Homomorphisms Between Structures In the developments that follow, we will need to find morphisms between different structures.

For example, we may need to find a homomorphism between two structures $S_1 = \langle A, B, L \rangle$ and $S_2 = \langle A', B', L' \rangle$, where $A, A', B, B' \in \text{TG}$ and L, L' are links of the form $\{E, (s, t)\}$.

This overall homomorphism will compose the homomorphism $f(A) = A', g(B) = B', h(L) = L'$, where f and g are the homomorphisms to be found, and h is the link homomorphism.

A reviewer pointed out that we also require that these sub-homomorphisms are consistent with each other. That is, the link homomorphism h must agree with the bindings given by f and g .

Bentley ► **Complete this** ◀

3 DSLTrans

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 ??.

3.1 DSLTrans Introduction

DSLTrans is a visual graph-based and rule-based model transformation engine that has two important properties enforced by construction: all its computations are both *terminating* and *confluent* [?]. These properties stem from the fact that DSLTrans does not allow unbounded loops during execution, making it a Turing-incomplete computing language [?]. Besides their obvious importance in practice, *termination* and *confluence*

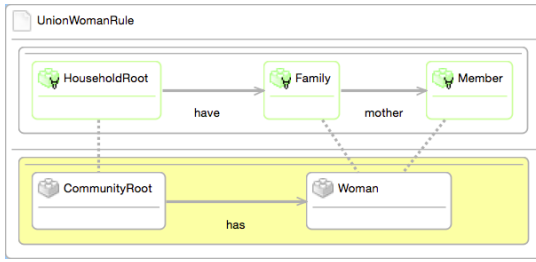


Fig. 1 An example of a DSLTrans rule

were instrumental in the implementation of our verification technique for pre-/post-condition contracts.

Model transformations are expressed in DSLTrans as sets of graph rewriting rules, having an upper part (named *MatchModel*), a lower part (*ApplyModel*) and, optionally, negative application conditions. The main construction used in the scheduling of model transformation rules in DSLTrans is a *layer*. Each model transformation rule in a layer cannot match over the output of any other rule in the same layer. As well, rules cannot modify the input graph during the rewriting phase (termed *out-place* execution). Layers are organized sequentially and the output model that results from executing a given layer is passed as input to the next layer in the sequence.

A DSLTrans rule can match over the elements of the input model of the transformation and also over elements that have been generated so far in the output model. Matching over elements of the output model of a transformation is achieved using a DSLTrans construct called *backward links*. Backward links allow matching over traces between elements in the input and the output models of the transformation. These traces are explicitly built by the DSLTrans transformation engine during rule execution.

For example, we depict in Figure 1 a rule in the DSLTrans language. When a rule is executed, the graph in the *MatchModel* of the rule is searched for in the transformation’s input model, together with the classes in the *ApplyModel* of the rule that are connected to *backward links*. An example of a *backward link* can be observed in Figure 1 as a dotted line connecting the *Country* and the *Community* match classes. During the rewrite part of rule application, the instances of classes in the *ApplyModel* of the rule that are not connected to backward links, together with their adjacent relations, are created in the output model.

For example, the *UnionWomanRule* rule in Figure 1 will match over a *Country* element connected to a *Family* element connected to a *Parent* element. If these elements are found in the input model along with the corresponding *Community* and

Woman elements in the output model, then a *persons* relation will be created between those output elements.

Although not present in this rule, copying object attribute values from the *MatchModel* to the *ApplyModel* of the rules is also part of the DSLTrans language, as illustrated in Section 3.5.

Reviewer ▶ *Please explain, earlier on, the meaning of indirect edges.* ◀

A DSLTrans transformation has a source and a target meta-model, which are seen in ???. This *Police Station* transformation will be presented throughout the rest of this paper as an example transformation. The metamodel in ??? 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 ??? 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 3.2. 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 will always 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.

Besides the fact that DSLTrans’ transformations are free of constructs that imply unbounded recursion or non-determinism, DSLTrans’ transformations are strictly out-place, meaning no changes are allowed to the input model. However, the output metamodel for a DSLTrans transformation can be the same as the input metamodel. Also, elements cannot be removed from the output metamodel as the result of applying a DSLTrans rule. This restriction is consistent with the usage of model transformations as translations [17], as no deletion of output elements is strictly required.

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

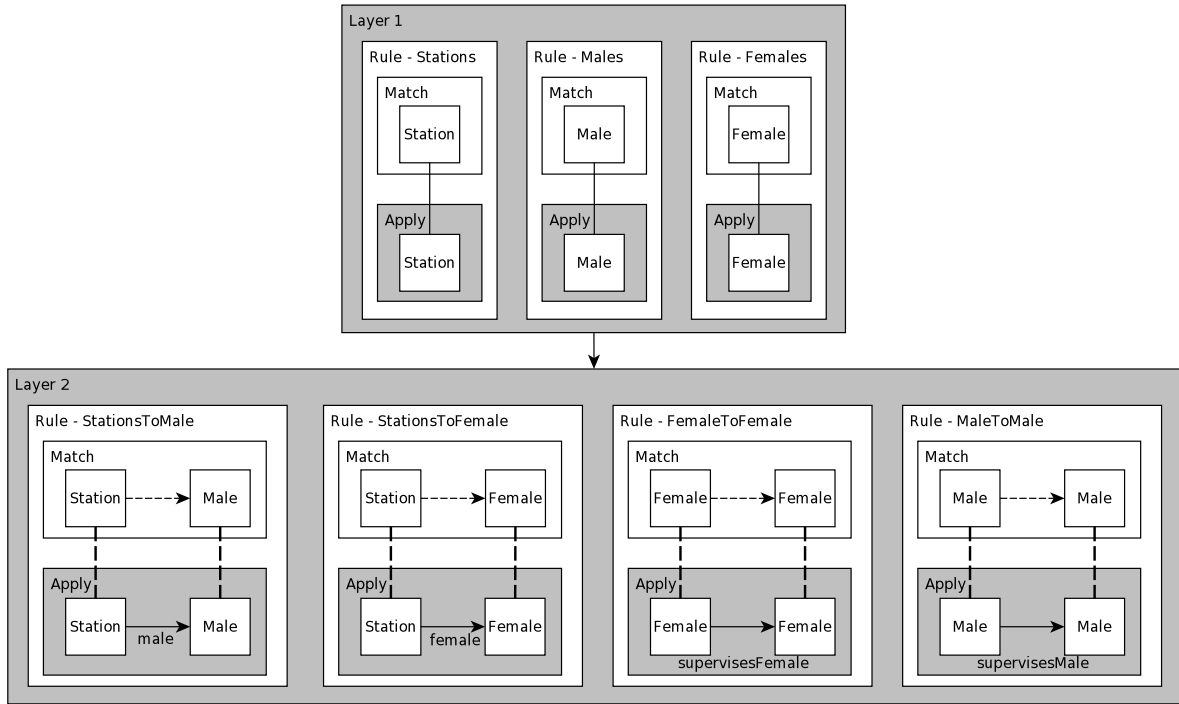


Fig. 2 The *Police Station* model transformation expressed in DSLTrans

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 mean some relations present in the input model will not be retained in the output model.

Reviewer ▶ **Figure 3: I would suggest adding another node *f5* supervised by *f4*, to illustrate the notion of transitive closure. Another remark: I think the black containment diamonds do not belong in an instance model, as they are a metamodel notion.** ◀

An example of this transformation's execution can be observed in ??, 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 ?? are instances of the source *Organization* metamodel elements *Station*, *Male* and *Female* respectively. The primed elements in ?? are their counterpart instances in the target *Gender* metamodel.

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 9 in Section 2.

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*, an 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 subgraph of the match graph of any other rule (as formally stated in ?? of model transformation, in Section 2 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 **Reviewer** ▶ **Perhaps 'the expressiveness of the rule language'?** ◀ is not restricted. In that work, we detail an operational rule processing step to handle overlapping rules.

Reviewer ► *In page 10 (answer to the reviewers) you say that rules for a layer must be parallel independent. Is this written anywhere in the main part of the paper?* ◀

In addition to the constructs presented in the example in Figure 1, DSLTrans has several others: *existential matching* which allows selecting only one result when a match class of a rule matches an input model, *indirect links* for transitive matching over containment relations in the input model, and *negative application conditions* that allow the transformation designer to specify conditions under which a rule should not match. These constructs are not currently used in our verification approach, and the interested reader is referred to [?] for further information.

3.2 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 and ??, while Section 3.7 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. **Reviewer** ► *a nonempty path (I think). The label of the containment associations does not matter?* ◀

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.

3.3 Families To Persons

As our running example we present an extended version of the *Families-to-Persons* transformation described in [?]. The original *Families-to-Persons* transformation can be found in

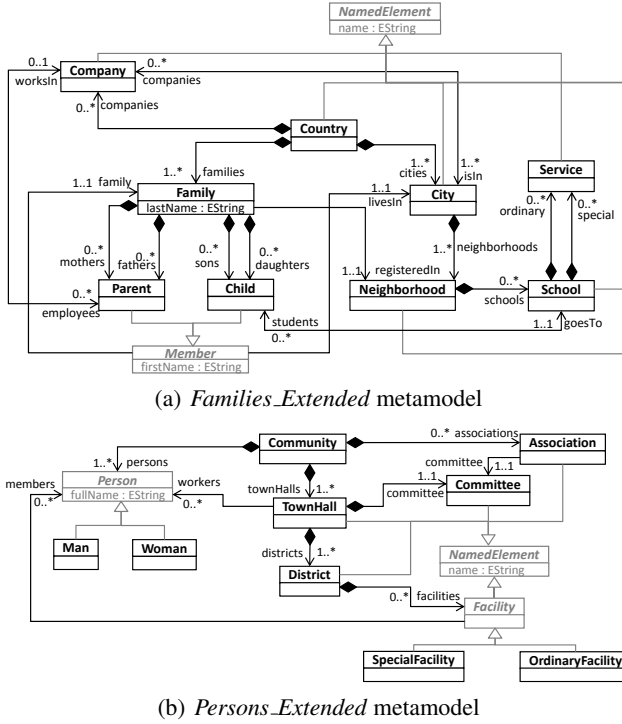


Fig. 3 Metamodels of the *Families-to-Persons_Extended* transformation

the ATL zoo [?], and has also been discussed in a number of related works on verification and testing [?].

We chose this *Families-to-Persons* transformation as our running example for two reasons. First, it transforms domains whose concepts are easily understandable by anyone (cf. Section 3.4). Second, it has a certain degree of complexity since it uses many features available in the ATL language (cf. Section ??).

3.4 Transformation Domains

The input and output metamodels of this transformation are shown in Figure 3. Please note that abstract classes are depicted in grey color and with italic names, and inheritance relationships are depicted in grey.

The input metamodel, the *Families_Extended* metamodel, has the *Country* class as a root element. A *Country* is made up of *companies*, *families* and *cities*. A *Family* has a *lastName*, is *registeredIn* a *Neighborhood* and can have any number of *mothers* and *fathers*, who are *Parents* and may, in turn, work in (*worksIn*) a *Company*. It can also contain any number of *sons* and *daughters*, who are *Children*, and every child *goesTo* a *School*. Both parents and children are *Members* that have a *firstName*, belong to a *family* and each of them *livesIn* a *City*.

A *City* may contain *companies*, and a *Company*, in turn, can be present in (relationship *isIn*) several distinct cities. A *City* is composed of *neighborhoods*, and these can have *schools*, where several *students* are registered. Every *School* has *Services*, and these may be *special*, for students with special needs, or simply offer *ordinary* services. Finally, countries, cities, companies, neighborhoods and schools have a *name* attribute, which is inherited from the abstract *NamedElement* class.

The output metamodel, *Persons_Extended*, is shown in Figure 3(b). The root class is *Community*, which is made up of *persons*, *townHalls* and *associations*. A *Person* has a *fullName* and can either be a *Man* or a *Woman*. An *Association* has a *Committee* that makes decisions. Every *TownHall* has a roster of *workers* (all the persons that are employed), hosts a *Committee* to make decisions, and also governs several *districts*. A *District* may contain several *facilities*, either of type *SpecialFacility* for those with special needs, or *OrdinaryFacility*. Each *Facility* may have registered several persons as *members*. Finally, associations, town halls, committees, districts and facilities have a *name* attribute.

3.5 DSLTrans Representation

Figure 4 displays the DSLTrans transformation which corresponds to the ATL *Families-to-Persons_Extended* transformation shown in Listing ???. Let us mention here that we have removed five rules from the figure to improve visual clarity. There is a vertical dotted blue line for each of these rules, located where the rules have been removed. The missed rules are similar to those that surround them, and can therefore be safely ignored in our explanation.

The process of constructing a DSLTrans transformation from an ATL one is described in the next section. For now, note that DSLTrans transformation obtained from ATL through the higher-order transformation includes only one rule per layer, meaning all rules execute sequentially. This is due to the sequential semantics of ATL that we replicate in DSLTrans.

Also, note that attribute copies are represented with arrows from the *ApplyModel* of the rule to the *MatchModel*, such as in rule *Neighborhood2District*, where the created *District* gets the same name as the matched *Neighborhood*. The string of an attribute of a created element can also be initialized with the concatenation of several strings. For instance, in rule *Father2Man*, the full name of the created *Man* comes

Fig. 4 DSLTrans version of the *Families-to-Persons_Extended* transformation

from the concatenation of the first name of the matched *Parent* and the last name of his/her *Family*. Or it can be assigned the string of an attribute of an element in the *MatchModel* concatenated with a given string, such as in rule *City2TownHall*.

3.6 Formal Structures

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

As discussed in Section ??, DSLTrans transformations are composed of rules arranged in layers.

3.6.1 Similar Structures Note that in the definitions that follow, in this section and others, that we define structures which are very similar in composition, which each containing two or three typed graphs, as well as a link component. This similarity is essential to our technique, as we will define morphisms between the various components of these constructions, which vary depending on the particular structure under examination.

While this composition of typed graph approach is preferred by the authors due to its reflection in our prover implementation, some readers may disagree. In this case, note that it is equivalent to think of these structures as one large typed graph annotated by the component it originates from, with appropriate projections for selecting relevant portions.

DSLTrans Transformation Rule A transformation rule is the elemental block of a DSLTrans transformation. Several transformation rules can be observed in the Police Station transformation in Figure 2.

A transformation rule includes a non-empty match pattern and a non-empty apply pattern. This is also known in the model transformation literature as a rule's *left hand side* and *right hand side*. As described in Section 3.4 or Section 2, which the match graph elements are found in the input model, then the apply graph elements are produced in the output model.

A match pattern can include indirect links that are used to transitively match containment relations in a model. 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. An apply pattern does not include indirect links as it is used only for the construction of parts of instances of a metamodel.

A rule may also contain a negative application condition (NAC) which if matched over the target graph prevents application of the rule.

A transformation rule also includes backward links, as informally introduced in Section 3.2. As described in that section, backward links define dependencies between rules.

Bentley ► **Expand** ◀

Definition 9 DSLTrans Transformation Rule

A DSLTrans transformation rule is a four-tuple $\langle NAC, Match, Apply, backward \rangle$, where:

- $NAC, Match, Apply \in TG$
- *Match and Apply are non-empty and are disjoint*
- *NAC may be an empty graph, and is disjoint from both Match and Apply*

Note that when we require an element of the Match, Apply, or NAC graphs, such as the vertices, we will index the required element. For example, the vertices for the Match graph will be V_{Match} .

- $backward = \{E_{back}, (s_{back}, t_{back})\}$
- E_{back} contains the backward links
 - E_{back} is disjoint from E_{Match} , E_{Apply} , and E_{NAC}
- (s_{back}, t_{back}) is a pair of functions $s_{back} : E_{back} \rightarrow V_{Apply}$ and $t_{back} : E_{back} \rightarrow V_{Match}$ that respectively provide the pattern and target vertices for each backward link
 - *Note the source of backward links is a vertex in V_{Apply} while the target is a vertex in V_{Match}*

$RULES$ is the set of all rules.

We additionally impose that for the well-formedness of a DSLTrans rule, there always exists an element or edge to be created in the *Apply* graph to be created. That is, either there are edges to be created in the *Apply* graph, or there is a vertex in the *Apply* graph that is not the source of a backward link.

- $E_{Apply} \neq \emptyset \vee$
- $\exists v \in V_{Apply} | \{ \forall e \in E_{back} : s_{back}(e) \neq v \}$

DSLTrans Layer and Transformation Definition 10 and Definition 11 formalise the abstract syntax of a model transformation, introduced in ??. An example of a model transformation can be observed in Figure 2, the Police Station transformation. As expected, a DSLTrans transformation is composed of a sequence of layers where each layer is composed of a set of rules.

Definition 10 Layer

A layer is a finite set of transformation rules: $\{r_0, r_1, \dots, r_n | r_i \in \text{RULES}\}$.

The set of all layers is denoted **LAYERS**.

Note that the order of the rules within the layer does not matter, due to the semantics of DSLTrans rule execution. As discussed in Section BLAH, these semantics ensure that rules are independent, and cannot act on the output of other rules within the same layer.

Definition 11 DSLTrans Transformation

A DSLTrans transformation is a finite list of layers denoted $\{l_0, l_1, \dots, l_n | l \in \text{LAYERS}\}$. Note that the order of layers in a transformation is important.

The set of all transformations is denoted **TRANSFORMS**.

Input-Output Model To describe the semantics of a DSLTrans model transformation, we must define an *input-output model* construct. This input-output model allows the representation of the initial state as well as intermediate operational states during the execution of a model transformation. These input-output models are thus the ‘path conditions’ which represent all transformation executions through an abstraction relation. This is further discussed in Section ABSTRACTION RELATION.

The structure of an input-output model is intentionally very similar to a DSLTrans rule. There is one typed graph representing the input model and another typed graph representing the output model. As well, the construct contains a set of edges, named *traceability links*, for keeping a history of which elements in the output model originated from which elements in the input model.



Definition 12 Input-Output Model

An input-output model rule is a three-tuple $\langle \text{Input}, \text{Output}, \text{trace} \rangle$, where:

- $\text{Input}, \text{Output} \in \text{TG}$
- Input and Output may be empty and are disjoint
- $\text{trace} = \{E_{\text{trace}}, (s_{\text{trace}}, t_{\text{trace}})\}$
 - E_{trace} contains the traceability links
 - E_{trace} is disjoint from E_{Input} and E_{Output}
 - $(s_{\text{trace}}, t_{\text{trace}})$ is a pair of functions $s_{\text{trace}} : E_{\text{trace}} \rightarrow V_{\text{Output}}$ and $t_{\text{trace}} : E_{\text{trace}} \rightarrow V_{\text{Input}}$ that respectively provide the source and target vertices for each traceability link

Let **IOM** be the set of all input-output models.

We define a utility function $\text{getTransformation} : \text{IOM} \rightarrow \text{Transforms}$. This function returns the Transform that the input-output model was built for. The purpose of this function is to restrict input-output models to only be applicable for the transformation they represent.

Note that the VT and ET for input and output will come from the input and output metamodels.  

3.7 Transformation Semantics

This section will discuss the semantics of a DSLTrans transformation. Given an input model and a transformation, an output layer will be produced through the repeated execution of rules and layers.

Execution of a DSLTrans Rule We will now address the execution of a rule in the DSLTrans language, basing our explanation on double pushouts.

Note that we will apply rules differently, depending on whether they contain indirect links or not. We will begin with the simpler case where the rule does not contain indirect links, diagrammed in Figure 5.

We create first the matcher and the rewriter for a DSLTrans rule.

Matcher of a Transformation Rule To create the actual matcher construct that will be matched during a rule’s execution, the rule’s *Match* graph is combined with backward links, as well as any *Apply* vertices connected to the backward links.

Note that the rule’s NAC will also be considered in the matching of a rule.

Definition 13 Matcher of a Transformation Rule

Let the transformation rule $r = \langle \text{NAC}, \text{Match}, \text{Apply}, \text{backward} \rangle$.

We define r ’s matcher, noted $\lceil r \rceil$, to be a typed graph six-tuple $\langle V, E, (s, t), \tau, VT, ET \rangle$, where:

- $V = V_{\text{Match}} \cup \{v \in V_{\text{Apply}} | \exists e \in E_{\text{back}} | s_{\text{back}}(e) = v\}$
- $E = E_{\text{Match}} \cup E_{\text{back}}$
- $s = s_{\text{Match}} \cup s_{\text{back}}, t = t_{\text{Match}} \cup t_{\text{back}}$
- $\tau = \tau_{\text{Match}} \cup \tau_{\text{Apply}}$
- $VT = VT_{\text{Match}} \cup VT_{\text{Apply}}$
- $ET = ET_{\text{Match}} \cup ET_{\text{Apply}}$

Rewriter of a Transformation Rule To continue with the double pushout approach, we must construct the rewriting (or replacement) graph.

The construction of this rewriter is essentially the same as the underlying rule. However, to support traceability, we require that backward links in the rule be converted into traceability links, and that new traceability links be created between all match and apply vertices.

Definition 14 *Rewriter of a Transformation Rule*

Let the transformation rule $r = \langle \text{NAC}, \text{Match}, \text{Apply}, \text{backward} \rangle$.

We define r 's rewriter, noted $[r]$, to be a structure $\langle \text{Input}, \text{Output}, \text{trace} \rangle$, where:

- $\text{Input} = \text{Match}$
- $\text{Output} = \text{Apply}$

We then modify trace to contain the appropriate traceability links

- $E_{\text{trace}} = \exists e \in E_{\text{trace}} |$
- $\forall m \in V_{\text{Match}}, a \in V_{\text{Apply}} : s_{\text{trace}}(e) = a, t_{\text{trace}}(e) = m$

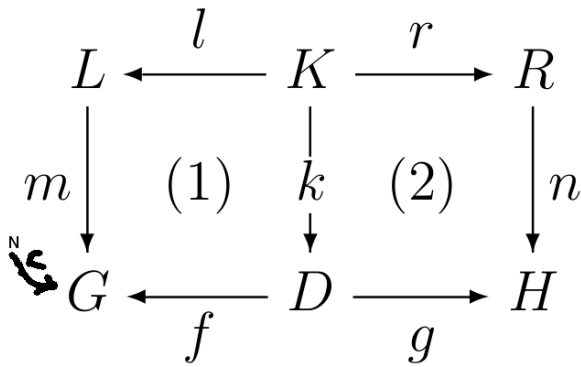


Fig. 5 Double pushout approach

Double-Pushout Approach If the reader is unfamiliar with the theory of double-pushouts, we refer to work on Fundamentals of Algebraic Graph Transformation.

The core idea of double-pushouts is to present graph rewriting in terms of morphisms, which compartmentalize the actions of finding a match and adding/deleting edges.

The application of a rule is called a *production*. Below, we will first explain the production of a single DSLTrans rule, expand this production to handle indirect links in the rule, then discuss multiple productions.

Let this production be for a rule $r = \langle \text{NAC}, [\text{Match}, \text{Apply}, \text{back}] \rangle$. To simplify the explanation, we constrain our rules to not include any indirect links. That is, $\nexists E \in E_{\text{Match}} \cup E_{\text{NAC}} | \text{isIndirect}(E)$.

Figure 5 presents a basic diagram for the approach. The major components are:

- L: The matcher of the rule $[r]$
- R: The rewriter of the rule $[r]$
- G: The graph to be matched/rewritten
- H: The rewritten graph

Rule Components K is the interface graph between L and R. That is, elements in K appear in both L and R. Elements which are to be deleted by the production appear in L but not K, and elements to be created are present in R but not K.

In the absence of indirect links, we note that $K = L$ as elements cannot be deleted in DSLTrans rules. Therefore the morphism l is isomorphism.

The morphism r from K to R is an injective typed graph homomorphism. Note that as described in Section 2.2.2, this homomorphism will be composed of sub-homomorphisms between the $\text{Match} \rightarrow \text{Input}$, $\text{Apply} \rightarrow \text{Output}$, and $\text{backward} \rightarrow \text{trace}$.

N represents the typed graph NAC of the rule. We note that this production cannot occur if there is a typed graph edge homomorphism c between the rule's NAC and G.

Graph Components G is the target graph which is to be matched on.

m is the typed graph homomorphism between the matcher of a rule r , denoted $([r])$ and the *Input* component of the input-output model.

D is the interface graph between the original graph G, and the rewritten graph H. Similar to K, D contains elements in both. Again, as DSLTrans rules do not delete elements, D will be isomorphic to G in this case, and f will be isomorphism. The morphism k will embed the K interface graph within the interface graph D.

The morphism g from the interface graph D to the rewritten graph H gives the graph elements, while the morphism r gives the embedding of the rewriter R with the elements to be produced. Thus, it is the pushout of R and D through K that gives H. g and n will be typed graph morphisms.

3.7.1 Double-Pushout Approach with Indirect Links If the DSLTrans rule contains indirect links, we must modify the

double-pushout approach described above to handle the transitive closure that the construct implies. The main change is to build the transitive edges, match on them, and then not build them in the final graph.

Note that this second explanation is valid for rules containing indirect links, or not containing them.

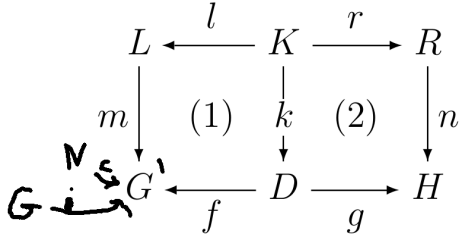


Fig. 6 Double pushout approach with indirect links

We add the component G' to the diagram, which represents the transitive closure of the Input graph of G . The homomorphism i gives the embedding of G into G' .

The homomorphism m will try to map the indirect links onto the newly created edges in G' .

K will not contain the indirect links. Therefore l will not longer be surjective, as nothing will map to the indirect links. R and r remain the same.

The interface D does not contain the edges newly created in G' . Thus the morphism f does not map onto these edges in G' .

n , g , and H remain the same.

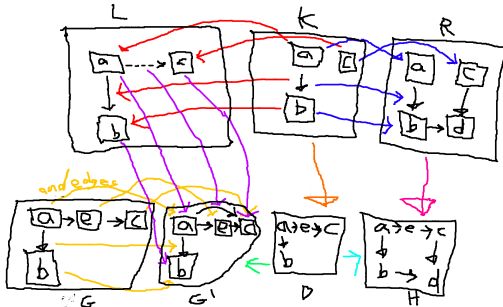


Fig. 7 DPO example with indirect links

Example Figure 7 shows an example of the DPO approach when the rule may contain indirect links.

Note how the orange homomorphism in the bottom left does not match to the newly created edges, nor does the green homomorphism from D .

The purple homomorphism from L does match the indirect link onto the created edges. But the red homomorphism does not match onto the indirect links in L , as they do not exist in K .

Bentley ► *Fix this up a lot.* ◀

DSLTrans Layer Execution Now that the production of a particular DSLTrans rule has been explained, the next definition defines how a DSLTrans transformation layer executes. Recall that a layer is composed of rules, and acts on the current input-output model.

Definition 15 DSLTrans Layer Execution

The execution of a DSLTrans layer can be defined as a function as follows:

$$\text{applyLayer} : \text{IOM} \times \text{Layer} \rightarrow \text{IOM}$$

$$\text{Let } l = \{r_0, r_1, \dots, r_n \mid r_i \in \text{RULES}\}, l \in \text{LAYERS}.$$

$$\text{Let } \text{IOM}_{\text{Input}}, \text{IOM}_{\text{Output}} \in \text{IOMS}.$$

$\text{IOM}_{\text{Output}}$ will be the result of applying the production p as defined below to $\text{IOM}_{\text{Input}}$.

We refer to the double-pushout literature for the creation of p . Specifically, we refer to the definition of parallel direct derivations. This p will be created out of the productions created for each of the rules r_i in the layer.

$$p = \langle (p_1, \text{in}^1), \dots, (p_k, \text{in}^k) \rangle : (L \xleftarrow{l} L \xrightarrow{r} R)$$

This collection of the rules is possible because DSLTrans rules do not delete anything. Essentially, a union is taken of the L , K , R components of each production.

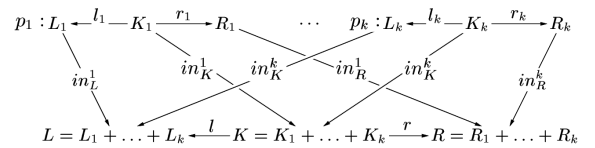


Fig. 8 Parallel production p

Definition 15 is the core of DSLTrans' semantics, in which we build the result of executing a layer of a DSLTrans transformation. 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 15 by the fact that the layer production is composed of a

union of the rule's productions. Rules belonging to the same layer are thus forced to execute independently, as described in the section on DSLTrans semantics.

Bentley ► *How does this handle different multiplicity of rule application?* ◀

DSLTrans Transformation Execution Our final definition is for the execution of a DSLTrans transformation. Essentially, it is the chaining of layer executions applied on an input-output model.

We will begin by discussing the conditions for executing a model transformation on an input model.

Input Conditions For a well-formed transformation execution, we require the *input-output model* in the domain to contain only an input graph. Recall that an *input-output model* is the three-tuple $\langle \text{Input}, \text{Output}, \text{trace} \rangle$. We therefore require that both the *Output* and *trace* components in the domain IOM to be empty. This input-output model thus represents the first step of the transformation, where no rule has been executed yet.

The input-output model in the co-domain will contain this input graph, as well as the output graph and traceability links produced by the execution of the transformation. Let EXECS_t be the set of all well-formed input-output models produced for transformation t .

Definition 16 *Execution of a DSLTrans Transformation*

The execution of a model transformation is a function $\text{applyTransformation} : \text{IOM} \times \text{Transforms} \rightarrow \text{IOM}$.

This function is a chaining of executions of those layers within the transformation.

Recall the application of a layer is $\text{applyLayer} : \text{IOM} \times \text{Layer} \rightarrow \text{IOM}$. Therefore, the application of a transformation is:

IOM as input to chaining of layer executions **Bentley** ► *Is this function chaining? I don't know how to write the equation for this.* ◀

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 16, the input-output model that is the output of executing a given layer is passed onto the next layer as input. The final input-output model created will be the result of the model transformation.

3.8 Confluence and Termination Properties

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 1 *Confluence*

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

Bentley ► *As we are relying on the double-pushout approach, I think we get this confluence for free.* ◀

Proof. We want to prove that for every model transformation execution of a transformation $\text{transform} \in \text{TRANSFORMS}$ having as input an input-output model $\text{input} \in \text{IOMs}$, 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. The section ‘Execution of a DSLTrans Rule’ discusses the matching and rewriting components to rule application. Note that our technique relies on the double-pushout approach, which itself is non-deterministic up to typed graph isomorphism, which does not contradict the proposition we are trying to prove. **Bentley** ► *Revise this.* ◀
2. In Definition 15, the rules composing a transformation layer are composed into a production on the input graph. Note that the order in which the transformation rules are composed may be non-deterministic. However, as these rules can be shown to be parallel independent **Bentley** ► *And they have to in that definition* ◀, the production is by definition confluent up to typed graph isomorphism.

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. \square

Proposition 2 *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 the matching and rewriting process of a rule is infinite. However, as the double-pushout approach uses a single production, then this must be a finite process. **Bentley** ▶ **Verify this, especially in the case of multiple applications of the rule.** ◀
2. if Definition 15 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 10.
3. if Definition 16 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 11.

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. ◻

4 Path Conditions and the Abstraction Relation

This section will present how path conditions are structured to represent symbolic rule execution. As well, 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.

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 meta-model 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 4.3.

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 ??, 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 9, 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.

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 4.3.

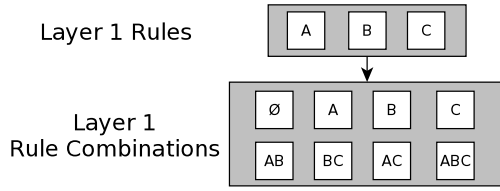


Fig. 9 Rule combinations created for a transformation layer

We also note that rule *combinations* are created, and not rule *permutations*. This follows from the semantics of DSLTrans as described in ??, 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 specify 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 ?. Note that symbolic traceability links are

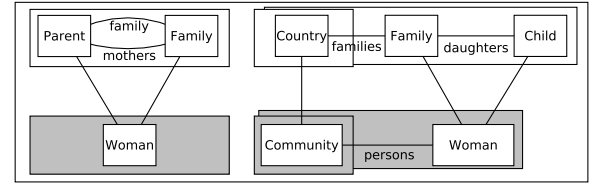


Fig. 10 An example path condition representing the execution of three rules

solid lines between match and apply elements in our visual notation.

4.2.2 Backward Links The dependencies in a DSLTrans rule are specified using the *backward link* construct, as further detailed in Section 3.2 and Definition 9. Section 5.2.2 will discuss how these dependencies are then resolved during our symbolic execution approach.

?? 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.

?? 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. Each path condition represents the execution of a set of transformation rules, by containing the input and output elements which are produced by the execution of those transformation rules.

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.

For example, the path condition in Figure 10 represents the execution of three rules in the transformation. This representation includes the input and output elements that will be present in the input and output models if these three rules execute. The set of path conditions produced by the prover will

therefore partition the set of valid executions of the transformation, where each execution is an input/output model pair. This technique was first proposed in [?] and further detailed in [?].

Definition 17 Path Condition

A path condition is a four-tuple $\langle \text{NAC}, \text{Input}, \text{Output}, \text{trace} \rangle$, where:

- $\text{NAC}, \text{Input}, \text{Output} \in \text{TG}$
- Input and Output may be empty and are disjoint
- NAC may be an empty graph, and is disjoint from both Input and Output
- $\text{trace} = \{E_{\text{trace}}, (s_{\text{trace}}, t_{\text{trace}})\}$
 - E_{trace} contains the traceability links
 - E_{trace} is disjoint from $E_{\text{Input}}, E_{\text{Output}}$, and E_{NAC}
 - $(s_{\text{trace}}, t_{\text{trace}})$ is a pair of functions $s_{\text{trace}} : E_{\text{trace}} \rightarrow V_{\text{Output}}$ and $t_{\text{trace}} : E_{\text{trace}} \rightarrow V_{\text{Input}}$ that respectively provide the source and target vertices for each traceability link

The empty path condition is defined as $\{\emptyset, \emptyset, \emptyset, \emptyset\}$.

The set of all path conditions is noted as PATHCONDS .

We also define a utility function $\text{getRule} : V \rightarrow \text{Rules}$. This function accepts a vertex from the path condition and returns the rule which created that vertex. This function will be used during the matching procedure of the abstraction relation, as elements from the same rule cannot match over the same target element. Note that this function allows production of the set of rules which are represented by the path condition. This information will be used to report the status of contract proof.

We define a utility function $\text{getTransformation} : \text{PC} \rightarrow \text{Transforms}$. This function returns the Transform that the path condition was built for. The purpose of this function is to restrict path condition to only be applicable for the transformation they represent.

The formal definition of a path condition is presented in Definition 17. Note that the structure of path conditions is intentionally very similar to that of DSLTrans rules and input-output models.

The *input* graph of a path condition represents a pattern that must be present in the input model of the transformation, while the *output* 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 *input* and *output* graphs to retain traceability information.

4.3 Abstraction Relation

Our abstraction relation is at the core of our technique. It allows us to represent an infinite set of transformation executions with a finite set of path conditions. As mentioned before, contract proof can then take place on this set of path conditions and hold on the set of transformation executions, as further explored in Section 7.2.

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

Definition 18 Abstraction of a Transformation Execution by a Path Condition

Let $\text{iom} = \langle \text{Input}, \text{Output}, \text{trace} \rangle \in \text{IOM}$ be an input-output model, also known as a transformation execution. Bentley ► Make sure this terminology is consistent. ◀

Let also $\text{pc} = \langle \text{NAC}, \text{Input}, \text{Output}, \text{trace} \rangle \in \text{PATHCONDS}$ be a path condition.

We also require that $\text{getTransformation}(\text{iom})$ and $\text{getTransformation}(\text{pc})$ return the same transformation.

We have that iom is abstracted by pc , noted $\text{ex} \Vdash \text{pc}$, if and only if the following conditions are all true:

Condition 1: The PC's NAC does not match onto the iom's Input

$$\nexists (PC_{\text{NAC}} \overset{\text{inj}}{\triangleleft} \text{iom}_{\text{Input}}^*) \quad (1)$$

Condition 2: The PC's Input matches onto the iom's Input. However, elements from the same rule in the PC cannot match over the same iom element. Note that this match function f is not injective.

The purpose of this relation is to enforce that the path condition accurately represents the elements present in the input model of the transformation execution. Note that extra elements may be present but not matched.

$$\exists (PC_{\text{Input}} \triangleleft \text{iom}_{\text{Input}}^* \text{ where } \forall v_1, v_2 \in V_{\text{PC-Input}} : f(v_1) = f(v_2) \vee \text{getRule}(v_1) \neq \text{getRule}(v_2)) \quad (2)$$

Condition 3: Elements in the iom's Output graph must surjectively match on the Output graph of the PC.

This surjection relation enforces that all elements which have been created in the output of the transformation execution must be represented by the output of the path condition. Note that this matching is from the execution to the path condition. As there may be multiple copies of an element in the execution but the path condition only represents execution of rules once, these elements should match over the same element in the path condition.

$$iom_{Output} \overset{surj}{\triangleleft} PC_{Output} \quad (3)$$

Condition 4: Traceability links in the PC must injectively match onto links in the transformation execution using the match function f . The purpose of this condition is to ensure that element creation represented in the path condition reflects actual rule execution present in the transformation execution.

Note that again we cannot have two traceability links from the same rule in the path condition matching over a traceability link in the transformation execution. Instead, both traceability links must be uniquely found.

TODO: Define the InjMatch we need here

$$PC_{trace} InjMatch iom_{trace} \quad (4)$$

Condition 5: Traceability links in the transformation execution must surjectively match onto links in the transformation execution. This case is to ensure that there is no element creation in the transformation execution that is not captured by the path condition.

TODO: Define the SurjMatch we need here

$$PC_{trace} SurjMatch iom_{trace} \quad (5)$$

Condition 6: The morphisms used in the above conditions must match.

TODO: Define how the morphisms agree.

$$PC_{trace} \quad (6)$$

4.4 Examples

In this section, we provide a number of examples to demonstrate the workings of the abstraction relation we chose to use. Figure 11 presents the legend for the following figures.

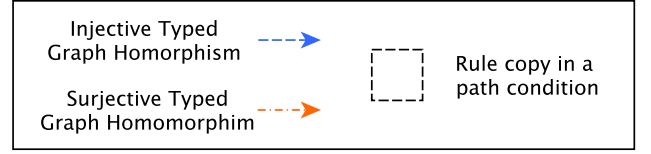


Fig. 11 Legend for abstraction relation figures

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

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 a transformation execution. Note that in both ?? and ??, 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 ?? and ??. 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 ?? 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 ?? 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.

4.4.2 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 transformation executions in ?? and ?. Note that this injection can be found in both cases.

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 ?, 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 ?? and ? 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.

4.4.3 Example 3 – Overlapping Rule Components For these examples, the path conditions contain overlapping rule com-

ponents, 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 ?? 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, ?? 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 ?. Each match is composed of a traceability link as well as immediately connected elements.

In contrast to ?, ? shows an example where the abstraction relation does not hold. Consider ?. 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 ?. 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.

4.4.4 Example 4 – Indirect Links We now present a path condition in ? 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 2 of Definition 18 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.

?? 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.

4.4.5 Example 5 – Combined Rules ?? 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 ?? is similar to other examples. The combined rules can be considered as a single graph for the abstraction relation.

?? 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 Building Path Conditions

In this section we present our approach to building a set of path conditions to represent all executions of a DSLTrans transformation.

5.1 Path Condition Generation Algorithm

?? 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 ??. 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 from Section 4.2 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 ??, 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 12 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.

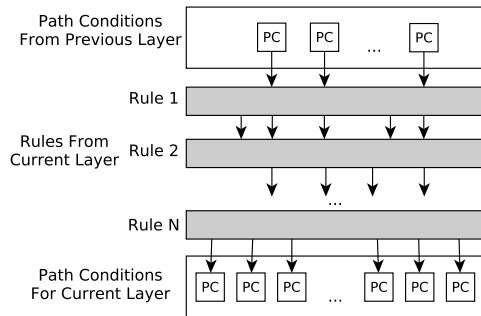


Fig. 12 Creating all path conditions for a layer

5.2 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*

Reviewer ▶ *A \sqcup operator is defined in Def. 19, but there you seem to rely on the fact that some elements of the path conditions and some elements of the rule are identical, otherwise the union would not work. Since a rule can be applied in several places, this would require a renaming of the rule elements, which - as far as I could see - is not specified.* ◀

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

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

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

Reviewer ▶ *Def. 19 can only be understood once it is clear that your plan is to make iso- morphic rule copies in which the node and edge identities already coincide with those in your path condition. (Hence my isomorphic representative guess in the previous remark.) This is very much nonstandard in the works of graph transformation, and poorly motivated. What are the advantages of this approach over relying on partial morphisms from a fixed rule to the path condition? As it is, you have to explain that you only consider isomorphic copies that differ in relevant parts, where relevant is determined by the question whether a node/edge identity coincides with one on the path condition. (You actually do not explain that at all: strictly following your definitions one would have to consider an infinite number of isomorphic rule copies, assuming that the number of available identities is infinite (as it must be).)* ◀

Reviewer ▶ *joint \rightarrow overlapping? Why are the type graphs actually not identical, as pc and rl are defined over the same source and target metamodels? Last line: $rl \rightarrow \{rl\}$* ◀

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 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 figures below 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.

5.2.1 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 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 13 and formally defined in Definition 20.

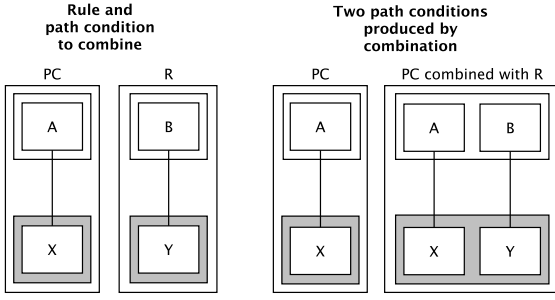


Fig. 13 R has no dependencies

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 $\xrightarrow{\text{combine}} \subseteq \text{PATHCOND}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr}) \times \text{RULE}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr})$ formally defined as follows:

$$rl = \langle V, E, st, \tau, Match, Apply \rangle, \nexists e \in E. \tau'(e) = trace$$

$$\langle pc, AC, rl \rangle \xrightarrow{\text{combine}} AC \cup \bigcup_{pc' \in AC} pc' \sqcup rl$$

Reviewer ▶ The right hand side of the conclusion of Def. 20 has $AC \cup_{pc' \in AC} pc'$, of which I cannot make sense. AC is a set, but the pc' are not, so what operation is $\bigcup_{pc' \in AC} pc'$? ◀

Reviewer ▶ Do you mean $\sqcup_{pc' \in AC}$ instead? But then the result is not a set, so how can you take its union with AC ? The same problem reoccurs in other definitions. ◀

Relation $\xrightarrow{\text{combine}}$ in Definition 20 models the operational combination step shown in ?? (the vertical black arrows between boxes). The relation has three input arguments: the first argument is the original path condition from the previous layer (shown as the topmost box in ?? 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.

5.2.2 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 14, 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

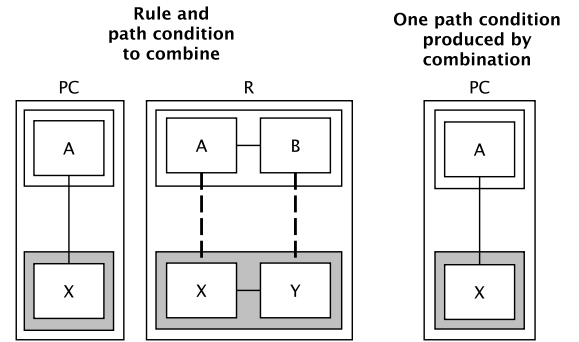


Fig. 14 R's dependencies are not satisfied by PC

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 $\xrightarrow{\text{combine}} \subseteq \text{PATHCOND}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr}) \times \text{RULE}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr})$, defined as follows:

$$\frac{\neg(rl|_{trace} \triangleleft pc|_{trace})}{\langle pc, AC, rl \rangle \xrightarrow{\text{combine}} AC}$$

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 by one single symbolic traceability link in the path condition Reviewer **► Could be rephrased** ◀. 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 ???. 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 ???. This "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 15, 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 15 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 16, 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 $\xrightarrow{p-comb}$ and $\xrightarrow{t-comb}$, both 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:

$$\frac{rl \cong rl_{glue} \sqcup ma_{\Delta}}{\langle AC, rl, rl_{glue} \rangle \xrightarrow{p-comb} AC \cup \bigcup_{pc \in AC} pc \sqcup (rl_{glue} \sqcup ma_{\Delta})} \quad (7)$$

$$\frac{rl \cong rl_{glue} \sqcup ma_{\Delta}}{\langle AC, rl, rl_{glue} \rangle \xrightarrow{t-comb} \bigcup_{pc \in AC} pc \sqcup (rl_{glue} \sqcup ma_{\Delta})} \quad (8)$$

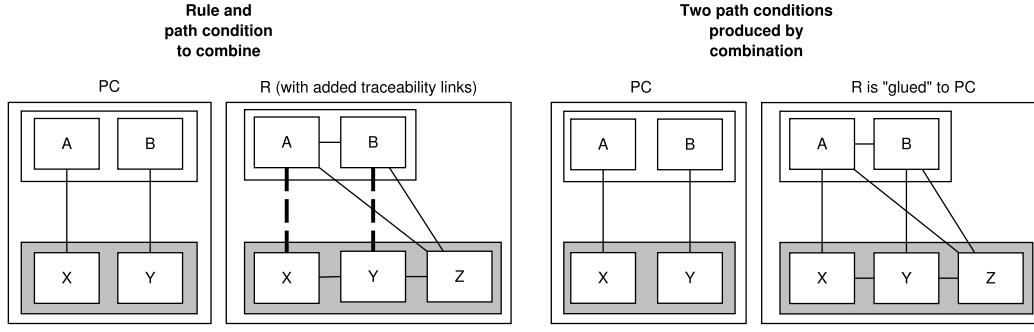


Fig. 15 R's dependencies are partially satisfied by PC

Let us start by introducing relation $\xrightarrow{p_comb}$, presented in Equation (7) 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 $\xrightarrow{t_comb}$ in Equation (8) 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 ?? and Figure 16. We formalize it in Definition 23, by means of relations $\xrightarrow{p_step}$ and $\xrightarrow{t_step}$. 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 $\xrightarrow{p_step}$ and $\xrightarrow{t_step}$, both 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:

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

$$\frac{rl_{glue} \in \text{partialSet}, \langle AC, rl, rl_{glue} \rangle \xrightarrow{p_comb} AC'', \quad \langle AC'', rl, \text{partialSet} \setminus \{rl_{glue}\} \rangle \xrightarrow{p_step} AC'}{\langle AC, rl, \text{partialSet} \rangle \xrightarrow{p_step} AC'} \quad (2)$$

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

$$\frac{rl_{glue} \in \text{totalSet}, \langle AC, rl, rl_{glue} \rangle \xrightarrow{t_comb} AC'', \quad \langle AC'', rl, \text{totalSet} \setminus \{rl_{glue}\} \rangle \xrightarrow{t_step} AC'}{\langle AC, rl, \text{totalSet} \rangle \xrightarrow{t_step} AC'} \quad (4)$$

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.

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

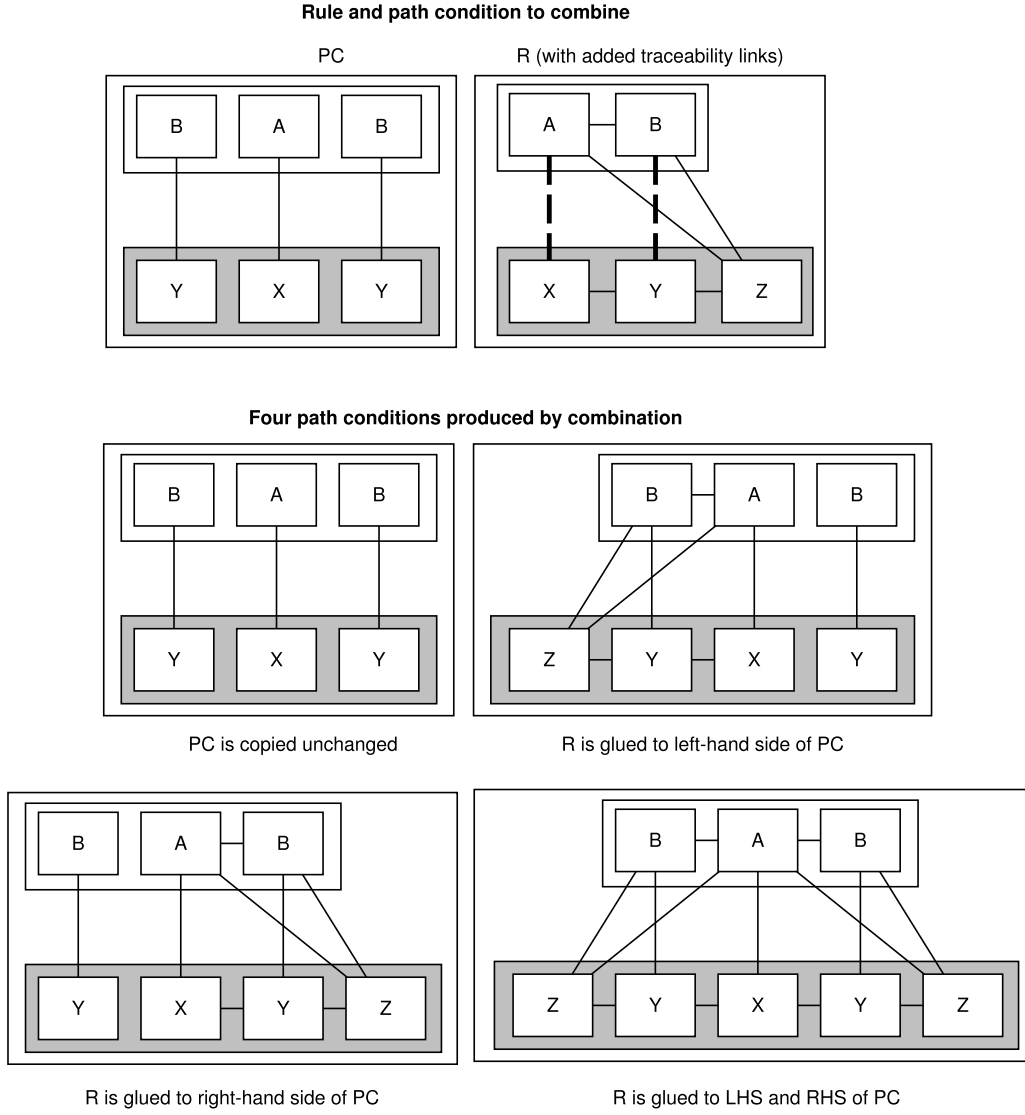


Fig. 16 R's dependencies are partially satisfied by PC, and are glued at all possible matches

and

$$\begin{array}{c}
 rl|_{\text{trace}} \triangleleft pc|_{\text{trace}} , \\
 \langle AC, rl, \text{partialsat}(rl, pc) \rangle \xrightarrow{p\text{-step}} AC'' , \\
 \langle AC'', rl, \text{totalsat}(rl, pc) \rangle \xrightarrow{t\text{-step}} AC' \\
 \hline
 \langle pc, AC, rl \rangle \xrightarrow{\text{combine}} AC'
 \end{array}$$

where

$$\begin{aligned}
 rl_{\text{glue}} \in \text{partialsat}(rl, pc) &\iff \\
 rl_{\text{glue}} \sqsubseteq pc^* \wedge rl|_{\text{trace}} \triangleleft rl_{\text{glue}} &\wedge \\
 \nexists rl'. (rl_{\text{glue}} \sqsubseteq rl' \sqsubseteq pc^* \wedge ||rl|| \triangleleft rl') &
 \end{aligned}$$

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

Reviewer ▶ **Def. 24:** *it seems to me you are making a very serious mistake here. ▶ tests for the existence of a morphism, but in Def. 19 you are relying on coincident node/edge identities. (See my remark 15.) Maybe you mean that the morphism from rl to pc should drive the choice of node/edge identity in the isomorphic copy of rl to be combined with pc, but this is really a wild guess on my part.* ◀

The top equation in Definition 24 defines the $\xrightarrow{\text{combine}}$ 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 $\xrightarrow{p\text{-comb}}$ and $\xrightarrow{t\text{-comb}}$ 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.

5.2.3 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 $\xrightarrow{\text{combine}}$ 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 12 and formalized in Definition 25 by the layer combination relation $\xrightarrow{\text{combpcsetlayer}}$.

Definition 25 *Combining a Path Condition with a Layer*

The layer combination relation $\xrightarrow{\text{combpcsetlayer}} \subseteq \text{PATHCOND}_{ig}^{sr} \times$

$\mathcal{P}(\text{PATHCOND}_{ig}^{sr}) \times \text{LAYER}_{ig}^{sr} \times \mathcal{P}(\text{PATHCOND}_{ig}^{sr})$ relation is defined as follows:

$$\frac{\frac{\frac{}{\langle pc, AC, \emptyset \rangle \xrightarrow{\text{combpcsetlayer}} AC} \quad rl \in layer, \langle pc, AC, rl \rangle \xrightarrow{\text{combine}} AC'', \quad \langle pc, AC'', layer \setminus \{rl\} \rangle \xrightarrow{\text{combpcsetlayer}} AC'}{\langle pc, AC, layer \rangle \xrightarrow{\text{combpcsetlayer}} AC''}}$$

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 $\xrightarrow{\text{combpcsetlayer}}$ in Definition 26.

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

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

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 4, the final set of created path conditions will represent every feasible transformation execution. Section 7 will discuss how our algorithm proves properties on these path conditions, and thus on all executions of the transformation.

Definition 27 *Path Condition Generation*

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

$$\frac{\frac{\frac{}{\langle AC, [] \rangle \xrightarrow{\text{pathcondgen}} AC} \quad \langle \epsilon_{pc}, layer^* \rangle \xrightarrow{\text{combpcsetlayer}} AC'', \langle AC'', tr \rangle \xrightarrow{\text{pathcondgen}} AC'}{\langle \epsilon_{pc}, [layer :: tr] \rangle \xrightarrow{\text{pathcondgen}} AC} \quad \text{where } layer^* = \bigcup_{rl \in l} rl^*$$

Note that in Definition 27, the recursive rule considers the expansion $(layer^*)$ of all the rules in a layer (see ??). 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 ??, the final set of created path conditions will represent every feasible transformation execution.

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

6 Validity and Completeness

This section 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.

Proposition 3 (Validity) *Every path condition abstracts at least one transformation execution.*

Proof sketch. Let $tr \in TRANSF_{tg}^{sr}$ be a DSLTans transformation. We wish to demonstrate that, for all path conditions $pc \in PATHCOND(tr)$, there exists a transformation execution $ex \in EXEC(tr)$ of the set of rules used to build pc such that pc abstracts ex , as formally expressed in Definition 18. We can prove this property by induction on the set of transformations $TRANSF_{tg}^{sr}$ (see ??), 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 2 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 all the conditions of the proposition. Proposition ?? 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.

- *Base case:* the base case is the case when we have $tr = []$, i.e. the empty transformation. In this case, according to ??, 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 ??), 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 (2) 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 all the conditions of the proposition; and b) ?? 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 transformation execution, we need to show that every path condition generated for a transformation tr' , resulting from adding a layer $l \in LAYER_{tg}^{sr}$ 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 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 $LAYER_{tg}^{sr}$.

- *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

Reviewer ▶ *three?* ◀ 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.

The property trivially holds for case ??, 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 ?? or ?? 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 ??. 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 $pc' = pc \sqcup^{trace} rl$, the combination of pc with rl as shown in ??. We first recall the conditions of the abstraction relation in ??:

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 ?? 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 ?? and ?? that one or more copies of graphs isomorphic to the apply part of rl are added to the output of ex . Also, by ??, 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 ?? 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 ?? 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 ?? 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 ??, replicates patterns that are produced in the execution by ?? and ??. 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 (??), 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 4 (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 18.

Completeness can be shown as a corollary of Proposition 3 about the *validity* of path condition generation. The complete set of executions $\text{EXEC}(tr)$ (see ??) 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 3, 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 3 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 3, 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.

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 ???. Completeness can be shown as a corollary of ??? about the *validity* of path condition generation. The complete set of executions $\text{EXEC}(tr)$ (see ???) 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 ???, 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 ??? 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 ???, ??? and ??? 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 ???, 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 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 ??? 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 (2) in Definition 18 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 ??? in ??? 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 ?? 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 ?? and ??, 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 ?? and ??, 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 ?? 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 ?? in ?? or the second part of Proposition ?? in Equation (2), 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 ??), 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 ?? in Equation (2) of the abstraction relation, it cannot be that either pc_1 or pc_2 abstract ex_1 and ex_2 simultaneously. \square

7 Verifying Properties of DSLTrans Transformations

The algorithm presented in Section 5 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 18 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 4 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.

7.1 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 28 *Property of a Transformation*

Reviewer ► **You can not handle properties with multiple instances of rule element (example in Fig. 25). On p.14 in the answer section, you claim that such conditions are excluded in Def. 29. However, the last lines of this definition are not easy to understand. Please explain why there are there (by giving the explanation from the answers section.) Without these explanations it is very hard to see why Prop. 3 should hold.** ◀

Let $tr \in \text{TRANSF}_{ig}^{sr}$ 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 $m \xrightarrow{f} Pre$ 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 28, 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 28 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 ??). 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 relevant² for the transformation at hand.

7.2 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 29 *Satisfaction of a Property by an Execution of a Transformation*

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

$$\forall f \exists g. (Pre \xrightarrow{f} Input^* \implies p \xrightarrow{g} ex^*)$$

$$\text{where } V(Input) \cap CoDom(g) = CoDom(f)$$

Definition 29 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

² In [6] we have referred to these properties *non-provable*. In the work presented here we explicitly disallow the construction of this class of properties.

property needs to be found in (the containment transitive closure of) the transformation execution. Note that the last part of the proposition in Definition 29 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.

?? 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 ?? 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.

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 30 *Satisfaction of a Property by a Path Condition*

Reviewer ▶ *In Def. 31 I found it confusing that - different from previous definitions - the two morphisms now go in the same direction. There is an explanation after Definition 31, but I found it very hard to understand.* ◀ 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 $pc = \langle V', E', st', \tau', Match, Apply, Rulecop \rangle \in \text{PATHCOND}(tr)$ be a path condition of tr . Path condition pc satisfies property p , written $pc \vdash p$, if and only if:

$$\forall f \exists g. (in \xleftarrow{f} Pre \implies out \xleftarrow{g} p) \\ \text{where } in \sqsubseteq Match^* \wedge out \sqsubseteq pc^*$$

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 30 is the same as the one behind the satisfaction relation between a property and an execution of a transformation in Definition 29: 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 29 and Definition 30. In Definition 29 – *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 30 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.

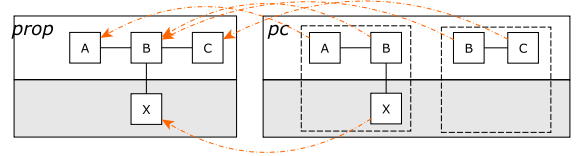


Fig. 17 Property satisfied by a path condition

This overlapping is modeled by the surjective typed graph homomorphisms of Definition 30 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 17. Note that element B is successfully matched even though it appears in two different rules in the path condition.

7.3 Expressiveness of the Property Language

As a result of taking rule combination (Section 5) and overlap (Section 7.2) 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 element of type A and that produces an element of type X. Our technique will create a path condition as seen in ??, 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 30, the property in ?? does not hold on the path condition in ??, although intuitively it should.

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 30.

7.4 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 ?? as Proposition 5 and ??.

Proposition 5 (Validity) *The result of proving a property on a set of path conditions generated for a transformation or on all executions of that transformation is the same.*

Let $tr \in \text{TRANSF}_{ig}^{st}$ 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:

$$\bigwedge_{pc \in \text{PATHCOND}(tr)} pc \vdash p \iff \bigwedge_{ex \in \text{EXEC}(tr)} ex \models p \quad (5)$$

Proof. In order to prove the proposition in Equation (5) 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 one execution ex of tr abstracted by pc . This lemma can be stated as follows:

$$pc \vdash p \iff \forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} . ex \models p \quad (6)$$

We thus need to demonstrate both directions of the equivalence in Equation (6). On the one hand we need to prove of the left-to-right direction of the equivalence:

$$pc \vdash p \implies \forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} . ex \models p \quad (7)$$

Proposition 7 is shown to be true in Lemma 2. We then need to show the right-to-left direction of the equivalence:

$$\forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} . ex \models p \implies pc \vdash p \quad (8)$$

Proposition 8 is shown to be true in Lemma 3. Once propositions 7 and 8 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 (5) is false, then according to Equation (6) 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 (6) is also false. \square

Lemma 2 *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:

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

Proof. By ?? we know that $pc \vdash p$ is equivalent to proposition $\forall f \exists g. (in \xrightarrow{f} Pre \implies out \xrightarrow{g} p)$, where Pre is p 's precondition, 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 ?? we also know that $ex \models p$ is equivalent $\forall f \exists g. (Pre \xrightarrow{f} Input^* \implies p \xrightarrow{g} ex^*)$, 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 9 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 :

$$\nexists f. (in \xrightarrow{f} Pre) \implies \nexists f'. Pre \xrightarrow{f'} Input^* \quad (10)$$

where, as before, $Input^*$ is the containment transitive closure of the input part of ex and in is a subgraph of the match part of pc . Proposition 10 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 ?? 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.

$$\forall f \exists g. (in \xrightarrow{f} Pre \wedge out \xrightarrow{g} p) \implies \nexists f'. (Pre \xrightarrow{f'} Input^*) \quad (11)$$

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 ?? and ?? we have that:

$$\begin{aligned} \forall f \exists g. (in \xrightarrow{f} Pre \wedge out \xrightarrow{g} p) \implies \\ \forall f' \exists g'. (Pre \xrightarrow{f'} Input^* \wedge p \xrightarrow{g'} ex^*) \\ \text{where } \text{Dom}(g) \cap \text{Match}(pc^*) = \text{Dom}(f) \text{ and} \\ V(Input) \cap \text{CoDom}(g') = \text{CoDom}(f') \quad (12) \end{aligned}$$

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 $out \xrightarrow{g} p \implies p \xrightarrow{g'} ex^*$. This proposition is true because we know by ?? 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 12 we know that in and out overlap on their subgraphs that are isomorphic to p 's precondition. By ?? 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. \square

Lemma 3 *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:

$$\forall ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} . ex \models p \implies pc \vdash p \quad (13)$$

Proof. We will demonstrate Proposition 13 holds by contraposing it:

$$\neg(pc \vdash p) \implies \exists ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} . \neg(ex \models p) \quad (14)$$

By ?? we know that $pc \vdash p$ is equivalent to proposition $\forall f \exists g . (in \xrightarrow{f} Pre \implies out \xrightarrow{g} p)$, 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 ?? that $ex \models p$ is equivalent $\forall f \exists g . (Pre \xrightarrow{f} Input^* \implies p \xrightarrow{g} ex^*)$, where $Input$ is the input part of ex . After replacing the left and the right hand side of Proposition 14 by equivalent formulas and solving the negations we reach the conclusion we need to prove:

$$\begin{aligned} \exists f \forall g . (in \xrightarrow{f} Pre \wedge \neg(out \xrightarrow{g} p)) &\implies \\ \exists ex \in \{ex \in \text{EXEC}(tr) \mid ex \Vdash pc\} . & \\ \exists f' \forall g' . (Pre \xrightarrow{f'} Input^* \implies \neg(p \xrightarrow{g'} ex^*)) &\quad (15) \end{aligned}$$

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 happens for at least one of the executions abstracted by that path condition. We know by Proposition 15 that $in \xrightarrow{f} Pre$, 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 $Pre \xrightarrow{f'} Input^*$ 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 ?? maps to the match part of pc onto $input^*$.

In order to complete the proof we need to show that the fact that $\neg(out \xrightarrow{g} p)$, i.e. if the complete property cannot be found in the path condition, then $\neg(p \xrightarrow{g'} ex^*)$, i.e. the complete property cannot be found in the execution. Note that, according to ?? and ??, 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 ??, 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 . \square

Proposition 6 (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 result follows from two previous results: Proposition 4, that tells us that every transformation execution is abstracted by one path condition; and ?? 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. \square

8 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.

8.1 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 5 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.

8.2 Complexity

Let us motivate our discussions of optimisation and performance by providing an approximate formula for the com-

plexity of the path condition construction and property proof algorithms presented in Sections 5 and 7.

8.2.1 Path Condition Generation Recall that a DSLTrans transformation is composed of rules arranged in layers. The path condition generation algorithm described in Section 5 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 5, 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.

$$O(2^r \cdot (m + g)) \quad (16)$$

Equation (16) presents the time and space complexity for the path condition generation algorithm.

8.2.2 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 7, 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 7.2. 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.

$$O(2^r \cdot 2^d \cdot p) \quad (17)$$

Equation (17) shows time and space complexity for the property proving step.

8.3 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 8.2.2 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 .

9 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.

9.1 Experimental Setup and Results

The complexity of Equation (16) and Equation (17) suggest that our property proving approach is intractable in the general case. However, we have provided in Section 8.3 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 ?? 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 18.

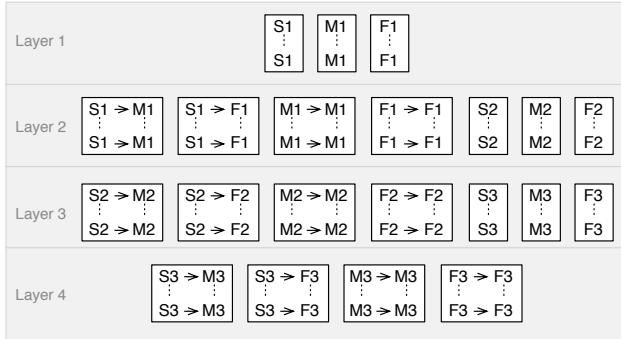


Fig. 18 Replicated Police Station transformation for performance tests

Note that for clarity reasons in Figure 18 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.

9.1.1 Results Reviewer **► Table 1, even if it was not experimented to exceed 0.003 seconds when proving for property that does not hold, maybe it would be possible to estimate the maximum time based on that measurement and the complexity formula.**◀

The results in Table 1 were obtained by verifying the properties in Figures ?? and ?? on the transformation seen in Figure 18. 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].

9.1.2 Time Required to Produce Path Conditions An important metric for our work is measuring how long it takes to produce the final set of path conditions from a DSLTrans transformation. As seen in Section 8.2, 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 18. 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.

?? presents the number of path conditions created for a given number of rules, while ?? 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 ?? 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.

9.1.3 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 ?? in ?. The first, in ?, is a property that we expect to hold for all path conditions. ? shows a property that we expect to *not* hold for all path conditions.

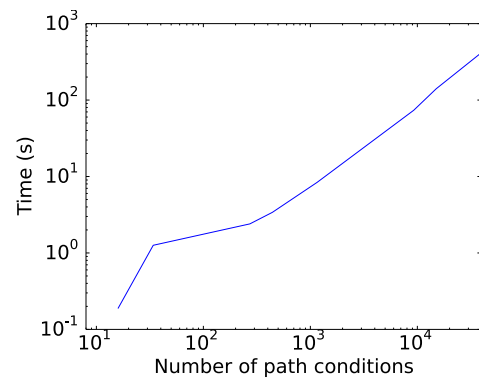


Fig. 19 Time required to prove the property that holds on all path conditions

Figure 19 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

# 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

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.

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.

9.2 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 ?? of Section 2. 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].

9.3 Discussion

The experimental results of verifying the test Police Station transformation portrayed in the graphs of ?? show that, as

predicted by complexity Equation (16), both the path condition construction time and the number of created path conditions grow exponentially with the number of rules. In Figure 19 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 8.2. 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 8.3. 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 9.2 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-world model transformations can vary within a wide range. The complexity of Equation (16) 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 (16) 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.

10 Related Work

Reviewer ► **citation [40]: Gonzalez and Cabot just published a novel paper at ICMT14.** ◀

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], AToM³ [25], VIA-TRA2 [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 *et al.* have published their work on the verification of ATL transformations [15,14]. In [15] the authors translate ATL transformations and their semantics into transformation models in Alloy. They then use Alloy's model finder to search for the negation of a given property that should hold, where the property is expressed as an OCL constraint. As the authors mention, Alloy performs bounded verification and as such it does not guarantee that a counterexample is found if it exists. In [14] Büttner *et al.* aim at proving model syntax relation properties of ATL transformations expressed as pre-condition/post-condition OCL constraints. In order to do so, the authors provide and use an axiomatisation of ATL's semantics in first order logic. Verification of a given model transformation is achieved by using a HOT to transform the transformation under analysis into additional first

order logic axioms. Off-the-shelf SMT solvers such as Z3 and Yices are then used to check whether the pre-condition/post-condition OCL constraints hold.

The approach in [14] comes very close to ours as the authors aim at proving the same type of properties in a model independent fashion and can do so exhaustively by using mathematical proofs at an appropriate level of abstraction, which can be seen as symbolic. However, there are several differences with our approach. First, the authors' proofs may require human assistance, depending on the used SAT solver. Also, despite the fact that Büttner *et 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' approach requires an intermediate logic representation of the transformation under analysis, our symbolic approach deals directly with transformation rules. This feature can ease the interpretation of analysis of results such as counterexamples and could be in general less error-prone due to the absence of an indirection layer which maps transformation concepts to concepts in the chosen logic. It is interesting to notice that, 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.

11 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 transformations where rules overlap in the match part (as per ?? in Section 2) 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 4 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 7.3.

From a practical standpoint, we have shown with the two examples presented in Section 9 that there are good indicators that our technique can scale to transformations of practical interest. We have shown in Section 8 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 7, 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

- Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software* **20** (September 2003) 42–45
- Mens, T., Van Gorp, P.: A Taxonomy of Model Transformations. *Electronic Notes in Theoretical Computer Science* **152** (March 2006) 125–142
- 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
- Lúcio, L., Barroca, B., Amaral, V.: A Technique for Automatic Validation of Model Transformations. In: *MODELS, Springer* (2010) 136–150
- 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>.
- Barroca, B., Lúcio, L., Amaral, V., Félix, R., Sousa, V.: DSLTrans: A Turing Incomplete Transformation Language. In: *SLE, Springer* (2010) 296–305
- 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>.
- Gomes, C., Barroca, B., Amaral, A.: DSLTrans User Manual <http://msdl.cs.mcgill.ca/people/levi/files/DSLTransManual.pdf>.
- 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>.
- Akehurst, D., Kent, S.: A Relational Approach to Defining Transformations in a Metamodel, Springer (2002) 243–258
- Narayanan, A., Karsai, G.: Verifying Model Transformations by Structural Correspondence. *Electronic Communications of the EASST* **10** (2008)
- Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL Contracts for the Verification of Model Transformations. *ECEASST* **24** (2009)
- 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
- Büttner, F., Egea, M., Cabot, J.: On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers. In: *MoDELS, Springer* (2012) 432–448
- Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL Transformations Using Transformation Models and Model Finders. In: *ICFEM, Springer* (2012) 198–213
- Asztalos, M., Lengyel, L., Levendovszky, T.: Towards Automated, Formal Verification of Model Transformations. In: *ICST, IEEE* (2010) 15–24
- 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)
- 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.
- King, J.: Symbolic Execution and Program Testing. *Communications of the ACM* **19**(7) (1976) 385–394
- Syriani, E., Vangheluwe, H.: De-/Re-constructing Model Transformation Languages. *ECEASST* **29** (2010)
- Syriani, E., Vangheluwe, H., LaShomb, B.: T-core: a framework for custom-built model transformation engines. *Software and Systems Modeling* (2013) 1–29
- Lúcio, L.: SyVOLT: A Prototype Implementation (2013) <http://msdl.cs.mcgill.ca/people/levi/files/SyVOLT.zip>.
- 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
- Taentzer, G.: AGG: A Tool Environment for Algebraic Graph Transformation. In: *AGTIVE. Volume 1779., Springer* (2000) 333–341
- De Lara, J., Vangheluwe, H.: ATOM³: A Tool for Multi-formalism and Meta-Modelling. In: *FASE '02, Springer-Verlag* (2002) 174–188
- Varró, D., Pataricza, A.: Generic and Meta-transformations for Model Transformation Engineering. In: *UML, Springer* (2004) 290–304
- Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. *Science of Computer Programming* **72**(12) (2008) 31 – 39
- 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
- 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 Definitions and Proofs of the Denotational Semantics

A.1 Basic Definitions

Definition 31 *Pattern* A pattern over a given set of vertex types $\mathcal{VT} = \{\mathcal{V}_1, \dots, \mathcal{V}_k\}$ and a given set of edge types $\mathcal{ET} = \{\mathcal{E}_1, \dots, \mathcal{E}_l\} \subseteq \{(\mathcal{V}_i \times \mathcal{V}_j \mid 1 \leq i, j \leq k)\}$ is defined as the structure $(V_1, \dots, V_m, E_1, \dots, E_n)$ with $V_i \in \mathcal{VT}$ for $i \leq i \leq m$ and $E_i \in \mathcal{ET}$ for $1 \leq i \leq n$. It represents an instantiable graph pattern by describing the vertices and edges used in that rule and the set of their possible instances.

Example 1 For the Stations Rule in Figure 2, the corresponding pattern is (S_O, S_G, T) where $S_O = \{\text{station}_{o,1}, \text{station}_{o,2}, \dots\}$ represents the set of all station instances from the organization language, $S_G = \{\text{station}_{g,1}, \text{station}_{g,2}, \dots\}$, the corresponding set of the gender language, and $T = S_O \times S_G$ the set of a trace links between a pair of instances.

Definition 32 *Instance Graph* An instance graph (V, E) with $V = \{v_1, \dots, v_m\}$ where $v_i \in V_i$ for $1 \leq i \leq m$ and $v_i \neq v_j$ for $i \neq j$, and $E = \{e_1, \dots, e_n\}$ where $e_i \in E_i$ for $1 \leq i \leq n$ and $e_i \neq e_j$ for $i \neq j$ of a pattern represents a graph instantiating a pattern, respecting the type constraints and multiplicities defined by that pattern.

Example 2 Again, using the Stations Rule in Figure 2, the set of corresponding instance graphs is $\{(\{s_o, s_g\}, \{(s_o, s_t)\}) \mid s_o \in S_O \text{ and } s_g \in S_G\}$. Thus, each instance graph consists of a pair of an organization station and a gender station, together with a single edge representing the trace link.

Definition 33 *Transformation Rule* A transformation rule over a given set of vertex types $\mathcal{VT} = \mathcal{VT}^i \cup \mathcal{VT}^o$ with $\mathcal{VT}^i \cap \mathcal{VT}^o = \text{emptyset}$ and a given set of edge types $\mathcal{ET} = \mathcal{ET}^i \cup \mathcal{ET}^o$ with $\mathcal{ET}^i \cap \mathcal{ET}^o = \emptyset$ is a partitioned pattern, describing a transformation by distinguishing between input elements \mathcal{VT}^i and \mathcal{ET}^i and output elements \mathcal{VT}^o and \mathcal{ET}^o .

A.2 Semantics of DSLTrans

Definition 34 *Set of Rule Executions.* Let $rl \in \text{RULE}_{ig}^{st}$ be a DSLTrans rule. The set of all rule executions is a set of graph relations, built inductively as follows:

$$\llbracket rl \rrbracket = \{(i', o') \in \text{GR} \mid (i', o') = (rl_\emptyset, rl_\emptyset) \vee (i', o') = (o, o \sqcup rl_{pat}) \wedge (i, o) \in \llbracket rl \rrbracket\}$$

where rl_{pat} is a pattern over rl and \sqcup is the tracing graph union where at least one element of $\llbracket rl_{pat} \rrbracket$ is disjoint with o .

A set of rule executions is the set of all input/output pairs resulting from executing one rule over any input model. It is built inductively by starting from rl_\emptyset (any input graph where the rule does not match) and adding to it any traced patterns of rl , meaning rl has executed any number of times. Note that the match part of rl_{pat} (denoted $\llbracket rl_{pat} \rrbracket$) can overlap with

the graph in o by at most $n - 1$ elements, being that n is the amount of match elements in rl_{pat} . Traceability adds trace links between any elements belonging all input elements and any output elements of rl_{pat} that is not connected to a backward link.

Levi ► **the definition of sets of rule executions is based on the notion of patterns over rules from Bernhard's text. I think I got the intent of 31, but more discussion is needed such that I'm sure of my definition. In particular the notion of overlap between graphs is underspecified regarding types (is this solved in the basic definitions?)** ◀

Definition 35 *Set of Rule Executions for a Layer.* Let $l \in \text{LAYER}_{ig}^{st}$ be a DSLTrans layer. The set of all layer executions is a set of graph relations, built inductively as follows:

$$\llbracket l \rrbracket = \begin{cases} \emptyset & \text{if } l = \emptyset \\ \llbracket l_1 \rrbracket \otimes \llbracket rl \rrbracket & \text{if } l = l_1 \cup rl \end{cases}$$

where $\text{EX}_1 \otimes \text{EX}_2$ is set of the unions of pairs of graphs $(ex_1, ex_2) \in \text{EX}_1 \times \text{EX}_2$ (EX_1 and EX_2 being sets of executions) where the match elements of ex_1 and the match elements of ex_2 may be joint.

Levi ► **this is overapproximating the set of layer executions because of the corner case where the match of one rule contains the match of the other rule in the layer, in which case the smaller rule will necessarily execute where the larger one does (but not vice-versa). I think there is an elegant way to describe this...or can we just use the overapproximation as is, given it does contain the set of all executions of the layer...** ◀