McGill University
School of Computer Science

Expressive Symbolic-Execution Contract Proving for the DSLTrans Transformation Language

Report No. 2017-01

Bentley James Oakes - bentley.oakes@mail.mcgill.ca
Levi Lúcio - lucio@fortiss.org
Cláudio Gomes - claudio.gomes@uantwerp.be
Hans Vangheluwe - Hans.Vangheluwe@uantwerp.be

September 15, 2017

Abstract

The verification of model transformations is key for the adoption of model-driven engineering in academic and industrial processes. In this work, we provide a verification technique for our model transformation language DSLTrans, which is both confluent and terminating by construction.

This technique proves structural pre-condition/ post-condition structural contracts for all inputs to a transformation. This is achieved by creating *path conditions* for the transformation through a symbolic execution of the transformation's rules. These path conditions then represent all possible transformation executions through an *abstraction relation*.

In this work, we provide a detailed description of both the path condition construction and contract proving techniques. As well, we provide arguments that our techniques are valid, such that proving a contract on the finite set of path conditions for a transformation implies that the contract holds on the infinite set of abstracted transformation executions.

1 Introduction

Model-driven engineering (MDE) is a process to manage the complexity in the systems under study by using models at an appropriate level of *abstraction*. One approach to managing complexity is through the creation of *domain-specific modelling languages*, where the modelling language encodes concepts from that domain to allow greater understanding by the domain expert [57, 58]. In the engineering of physical systems, MDE has been highly successful and widely accepted [56].

Model transformations are central for manipulating models in MDE [16, 54]. The reason for the adoption of transformations is their excellent compromise between strong theoretical foundations and applicability to real-world problems [42].

In particular, model transformations allow for mathematical treatment based on the foundations of graphs and graph transformations, while operating on the domain-specific concepts expressed in the *metamodels* (the model languages) for the transformation.

As seen in recent case studies, domain-specific transformations have a high level of abstraction, allowing for more effective collaboration between the transformation designer and the domain expert [57].

Transformations may perform a number of operations on a model, such as creating, updating, or deleting model elements [22, 28]. The literature also divides transformations into categories based on their intent [42], such as *simulation* to represent a system changing over time [23], or as *migration* to change the language of the model while preserving the content [53, 51].

Note that as the migration changes the langauge of the model, it is termed *exogenous*, while a transformation retaining the same language is termed *endogenous*. It should be noted that our research on model transformation verification focuses on translation or migration transformations.

Even though a transformation may be written at an appropriate level of abstraction within the appropriate domain, this does not imply the *correctness* of that transformation. As adoption of MDE increases in both academia and industry, the verification of these transformations becomes a critical part of software development.

For example, code generation transformations turn a software system model into executable code. These processes may be subject to industry standards such as the ISO-26262 [33] standard in the automotive domain, which requires documentation that the produced artifacts are correct.

To support this verification, it may be necessary to restrict the subset of the language being verified or to analyse the transformation for properties to hold [59]. In our research, we have instead constructed our verifiable model transformation language DSLTrans with the properties of *termination* and *confluence* [45].

1.1 Paper Contributions

This article will focus on the formalization of our verification technique for the model transformation language DSLTrans [11]. This work is based on our complete formalization of DSLTrans [45] which provides precise semantics for all constructs in DSLTrans.

We note that parts of our formalization are based on that from the PhD of Barroca [12], which also described a symbolic execution approach to contract proving. However, the approach in that work creates a combinational explosion when creating the symbolic state space for the transformation, rendering it impractical for industrial use.

Our previous technical report of Lúcio *et al.* [40] was an attempt to formalize the DSLTrans language and contract prover process. The contribution for that work was the creation of an abstraction relation such that only one copy of a rule would be present in a path condition. This allows the prover to scale up to reasonably-sized transformations. This article is an improvement on those earlier works by reformulating the contract proving process to employ a typed graph split morphism (Definition 5).

The use of this morphism allows for path conditions to represent multiple transformation executions without an explicit 'disambiguation' step. This disambiguation step was present in earlier research, and unified all possible combination of elements within rules. However, this is computationally infeasible as this operation explodes the state space.

As well, this work examines the critical issue of rule multiplicity in path conditions. As it is impossible to represent the repeated application of all rules within the transformation, it is necessary to abstract over the number of times each rule applies. We examine here how to explicitly represent each application of the rules, and the consequences for path condition construction and contract proving.

The current work also addresses DSLTrans constructs not considered by previous works([11, 39, 12]). Armed with the formalization work in Oakes *et al.* [45], we can now consider all DSLTrans constructs and therefore all DSLTrans transformations for verification.

The concrete contributions of our work are the following:

- A technique for constructing all path conditions representing all feasible executions of a DSLTrans transformation through an abstraction relation.
  - As an improvement to [40], a new and more intuitive explanation of path condition construction is provided which includes negative elements in rules and removal of invalid rule combinations
  - Discussion of a number of abstraction relation examples, including abstraction of multiple rule execution
  - Path conditions can now be built for all DSLTrans transformations as all DSLTrans constructs are now considered in our approach.

- A contract-proof algorithm that proves structural contracts over all path conditions, and therefore over all transformation executions. Counter-examples to these contracts are detected.
- Discussions of validity for the path condition construction and contract proof techniques.

We begin this paper by presenting an overview of the DSLTrans language in Section 2. Section 3 then briefly introduces the problem of proving contracts on a transformation and raises a number of the complications which we address in this paper. The structures of the language are then described in Section 5. To keep our focus on the proving of contracts in this work, we refer to the work of Oakes *et al.* [45] for the semantics of DSLTrans.

Following this, the remainder of the paper discusses our symbolic-execution verification of DSLTrans. Our approach symbolically executes rules from the transformation, relying on an abstraction relation defined in Section 6. Through resolution of dependencies, this procedure creates a set of *path conditions*, which indicate possible input and output models for the transformation (Section 7).

Structural contracts can then be proved on these path conditions, allowing the transformation verifier to understand the relationship between input and output models for the transformation. This contract verification is described in Section 9 with a discussion of validity presented in Section 10.

In Section 11 we present work related to the contract verification approach. Finally, we conclude this paper in Section 12.

2 DSLTrans Introduction

We present selections from the *Families-to-Persons* transformation from Oakes *et al.* [46] as an example DSLTrans transformation.

The input and output metamodels of this transformation are shown in Figure 1. Note that abstract classes are depicted in grey and with italic names, and inheritance relationships are depicted in grey.

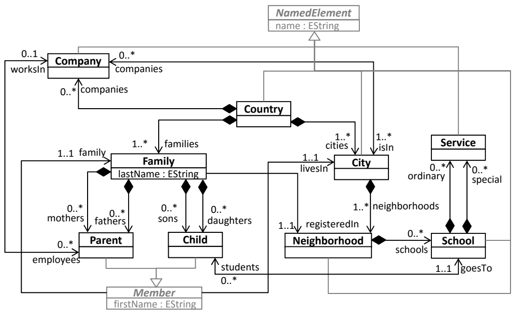
The original *Families-to-Persons* transformation is found in the Atlas Transformation Language (ATL) transformation zoo [1], and was selected because the domain concepts are simple to understand and it is representative of translation transformations. This transformation is also notable because it has been discussed in a number of related works on verification and testing [27].

Note that the DSLTrans version of this transformation was created automatically through a higher-order transformation applicable to all declarative ATL specifications [46].

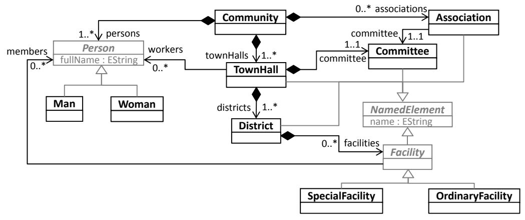
2.1 DSLTrans Rules

The essential unit of a DSLTrans transformation is the rule, containing a classical left-hand side (termed a *MatchModel*), and a right-hand side (termed an *ApplyModel*).

The semantics of rule execution, formally described in Oakes *et al.* [45], are to match the elements found in the *MatchModel* onto the input model. If this can be accomplished, then the elements in the *ApplyModel* are created in the output model.



(a) *Families_Extended* metamodel



(b) *Persons_Extended* metamodel

Figure 1: Metamodels of the *Families-to-Persons_Extended* transformation

For example, in Figure 2, if an element of type *Parent* is found connected to an element of type *Family* in the input model, then an element of *Man* is created in the output model.

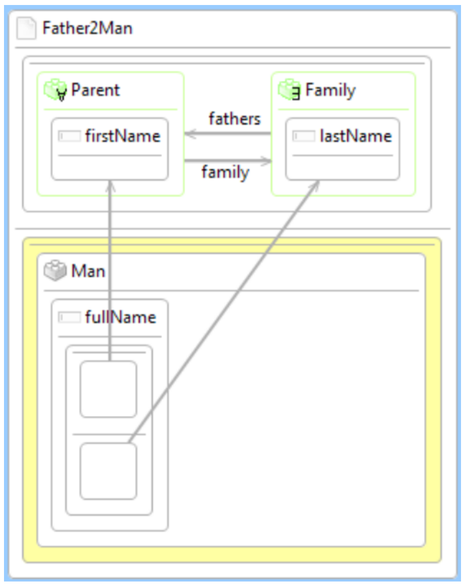


Figure 2: An example of a DSLTrans rule

Elements in both the *MatchModel* and *ApplyModel* components are typed by a meta-model, termed the source and target metamodel respectively. Note that if the source and target metamodel are the same, it is termed an *endogenous* transformation. When a transformation changes the language of the model, it is termed *exogenous*.

Also present in this rule example is the copying of attribute values from the *MatchModel* to the *ApplyModel*. In this example, the *fullName* of the *Man* is created from the concatenation of the *firstName* of the *Parent*, and the *lastName* of the *Family*.

2.2 DSLTrans Layers

The construction used for the scheduling of DSLTrans rules is a *layer*. The transformation is formed from layers where each layer contains a set of transformation rules. Layers in DSLTrans 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.

Note that while the transformation will execute layer-by-layer, rules in a layer will execute in a non-deterministic order. However, an essential property of the DSLTrans language is that each rule in a layer cannot match over the output of any other rule in the same layer, and rules can only modify the output graph during the rewriting phase of the rule's execution. This is termed *out-place* execution. As discussed in Section **??**, this is necessary for DSLTrans to be *confluent* by construction.

In Figure 3 we present the first four layers in a DSLTrans transformation which execute in sequence from left to right.

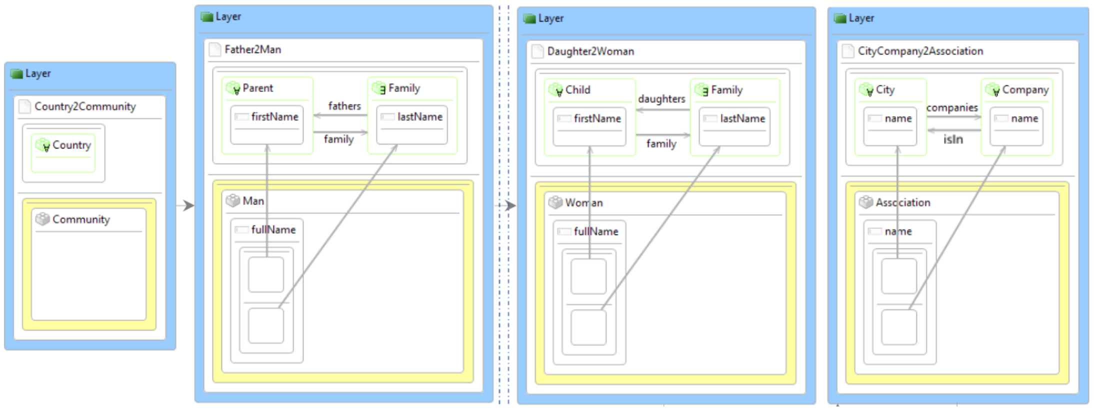


Figure 3: Four layers in a DSLTrans transformation

2.3 Backward Links

DSLTrans rules specify how elements of the output model were created from specific elements of the input model during rule execution. This traceability information for the transformation is stored as *traceability links*, which are built between a newly generated element in the output model, and the elements of the input model that originated it. For example, whenever the rule in Figure 2 applies, traceability links are created from the matched *Parent* and *Family* elements to the produced *Man* elements in the output model.

The *backward link* construct in DSLTrans rules is used to match over these traceability links. Figure 4 depicts a rule containing backward links. The backward link is denoted as a dotted line connecting the *Country* and *Community* elements. This link means that the *Community* element must have been created from that *Country* element in an earlier layer of the transformation.

When a rule is selected for application, the elements in the *MatchModel* of the rule are searched for in the transformation's input model, together with the elements in the *ApplyModel* of the rule that are directly connected to *backward links*. If these elements were found, then those *ApplyModel*

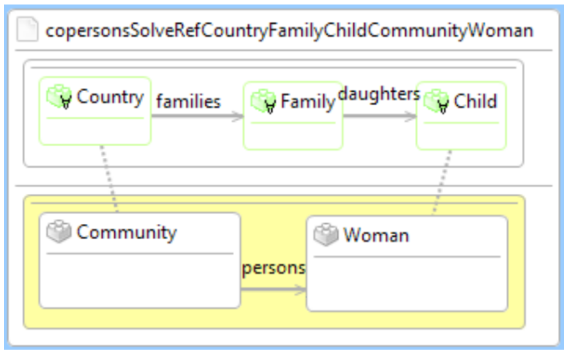


Figure 4: A DSLTrans rule with backward links

elements not connected to the backward link are created in the output mode during the rewrite part of rule application.

For example, the *copersons...* rule in Figure 4 will match over a *Country* element connected to a *Family* element connected to a *Child* 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.

2.4 DSLTrans Construct Reference

This section will briefly describe all of the constructs present in the DSLTrans language as a reference for the reader. Some of these constructs are seen in the transformation presented in Figure 3. Note that formal details for the syntax and semantics of these constructs are found in Oakes *et al.* [45].

A further description of the DSLTrans language (including installation instructions) is found in the DSLTrans manual [29].

- **Match Elements**: Match elements are contained in the left-hand side of rules, and represent elements which will match over all elements of the corresponding type (or subtype) in the input model when the transformation is executed. Note that the RAMification process [37] is used to create the matching elements, such that we can create a *School* match element to match over input elements of type *School*.

- **Existential Matching**: The *Exists* construct allows selecting at most one result when a match element matches onto an input model. These are denoted as *Exists* elements, as opposed to *Any* elements.

- **Direct Match Links**: Direct match links represent labelled associations typed by the source metamodel. These match links will match over associations of the same type in the input model.

- **Indirect Match Links**: Indirect match links are similar to direct match links, but there must exist a non-empty path without repeated elements between the matched elements for the indirect link to match.

  The typing of associations is ignored when matching indirect links.

- **Backward Links**: Backward links connect elements of the match and the apply patterns of a DSLTrans rule to represent dependencies on element creation by previous layers of the transformation. Backward links match over traceability links between elements of the transformation's input and output models.
- **Negative Elements**: Match elements, backward, and match links can all be labelled as *negative*. This prevents the matching of those elements onto the input model, allowing the transformation designer greater control over the execution of the transformation.
- **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. As previously mentioned, apply elements in a given transformation rule that are not connected to backward links will create elements in the output model. Apply links will always be created in the transformation's output model.

  During the execution of a DSLTrans rule, these output elements and links will be created as many times as the match graph of the rule is found in the input model.
- **Attributes:** DSLTrans includes a small language for matching over and creating *String* attributes:
  - *Atoms*: *String* literals.
  - *Wildcards*: Tokens to match over arbitrary values, similar to the star operator (Kleene star) in regular expressions. For example, *Wildcards* could be used to match attribute values that start with the *String "pre"*.
  - *Match Attribute References*: References to the value of match attributes. Labelled as *AttributeRef* in DSLTrans editors.
  - *Concat*: Used to concatenate atoms, wildcards, and attribute references.
- **Match Attributes**: Used to match attribute values contained in match elements, to restrict the application of rules.
- **Apply Attributes**: Apply attributes can be created from concatenating *String* literals or references to one or more match model element attributes.

3 Problem Overview

In this paper, we focus on the proving of structural contracts on DSLTrans transformations by resolving the interactions of transformation rules. This section will discuss the broad strokes of the problem and our approach, while more details about our contract proving technique are provided in Section 9.

Our contract proof technique has been successfully used in industrial applications [53, 51, 52]. These case studies show that the DSLTrans language can express useful 'translation' transformations, and then have their correctness verified by our contract proving approach.

We are also currently developing an industrial case study on the mbeddr development language which consists of 49 rules [6]. The mbeddr language is designed to easily create embeddable C code.

3.1 Structural Contract Proving

Our technique allows a proof that pre-/post-condition contracts hold on all input models for a given model transformation. These contracts represent patterns on the input and output models of a transformation, including traceability information. If a contract is said to hold by our technique, then a formal guarantee exists that whenever a transformation's input model contains the pattern specified in the pre-condition of the contract, the output model produced by that transformation will contain the pattern specified in the contract's post-condition.

For example, Figure 5 shows an example contract from the *Families-to-Persons* transformation. This contract will hold on the *Families-to-Persons* transformation if for all input models where there exists a *Country* containing a *City* and a particular *Company*, there must exist an associated *Community* with a *TownHall* and *Committee* in the output model.

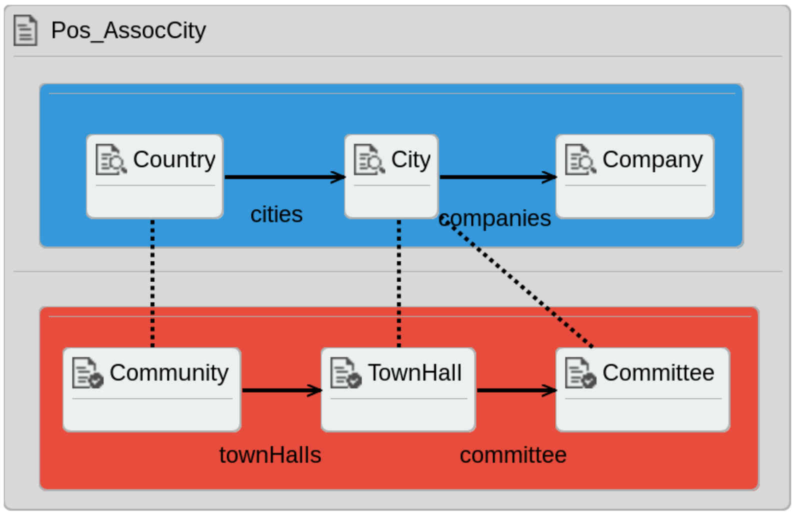


Figure 5: *Pos_AssocCity* contract

The contracts to be proved on DSLTrans transformation have an implication form. Similarly to rules and path conditions, contracts are composed of a pre-condition, a post-condition, and traceability links, as described in Section 9.1. Contracts thus represent the following statement: whenever the pre-condition is found in an input model to the transformation, then the post-condition (including traceability constraints) must be found in the corresponding output model.

For example, the contract in Figure 6a represents the informal statement "A *Family* with a *father*, *mother*, *son* and *daughter* should always produce two *Man* and two *Woman* elements in the target *Community*". Based on the *Families-to-Persons* transformation (Section 2), this contract should always hold over all transformation executions. The *Family* element should be transformed to a *Community* element, while each *Member* element should be transformed to the appropriate *Male* or *Female* element.

During our path condition creation, our technique builds structures which represents the execution of rules in the transformation, called *path conditions*. The input model of a path condition includes a representation of all the elements and associations created in the input model for a set of transformation executions. Likewise, the contract's pre-condition represents the required input elements for the property.

Thus, the first step in the contract proving algorithm is to examine whether the contract's pre-condition can be found in the path condition's input graph. If not found, then the contract will not be checked further on this path condition as the pre-condition does not hold.

On each path condition where the contract's pre-condition is found, the contracts's post-condition is searched for in the path condition. If the post-condition is also found, then all post-condition elements will be produced in all transformation execution(s) abstracted by that path condition. If the post-condition elements are not found, then the contract fails for that path condition.

Therefore, there are three cases for determining the status of a contract:

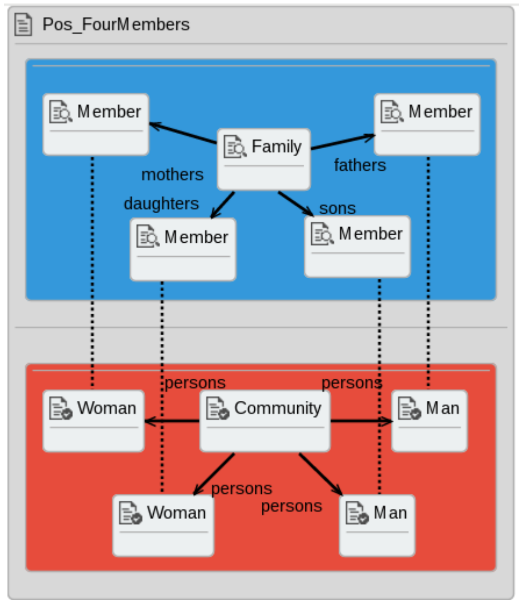
- If the pre-condition of the contract, including backward links, does not match the path condition, then the contract is not applicable for that path condition.
- If both the pre-condition and post-condition match, then the contract does hold on that path condition.
- If the pre-condition matches, but the post-condition does not match, then the contract does not hold on that path condition.

Note that a contract may be expected to not hold in all cases for a transformation. For example, consider the contract *DaughterMother* from the *Families-to-Persons* transformation [46], reproduced in Figure 6b. The informal statement for this contract is *'a family with a mother and a daughter will always produce a community with a man'*. It is easy to see that an input model which contains only mother and daughter elements should not produce a man in the target community.

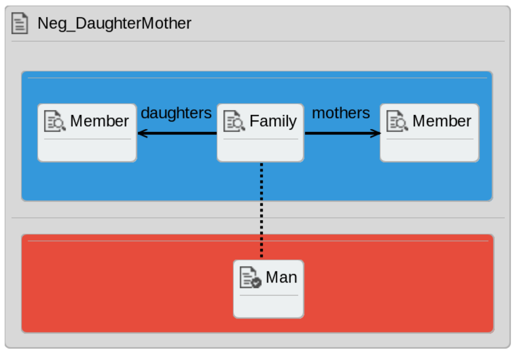
Our contract prover will then find multiple counter-example path conditions which cause the contract to not hold, such as the path condition in Figure 7. Note that the pre-condition of the contract does match onto the top component of the path condition, while the *Man* element in the contract post-condition cannot be found in the bottom component of the path condition. Thus the failure of this contract gives further assurance that the transformation is working correctly, as *daughters* and *mothers* are not accidentally transformed into *men*. As this result is expected, this allows the transformation builder to have increased confidence in the validity of the transformation.

If a contract fails to hold on the transformation and it was not expected to fail, then this indicates an error with either the contract or the transformation. Our contract-proving approach will detect which path conditions which the contract did not hold on. As well, a minimal path condition is reported by our implementation, which represents the smallest combination of rules that fails. This allows the transformation developer to identify those rule combinations wherein an error may occur, and change the transformation or contract accordingly.

Note that elements in the contract can also contain attributes to be matched over the transformation execution or path condition. As discussed in Oakes *et al.* [45], this attribute matching is subsumed as part of the morphism. As seen in [46], the *Families-to-Persons* transformation is verified using this attribute matching to ensure that the names of *Man* and *Woman* elements is correctly created from the appropriate *Member* elements.



(a) Contract Pos-FourMembers – Expected to hold



(b) Contract Neg-DaughterMother – Not expected to hold

Figure 6: Contracts to be proved on the Families-to-Persons transformation

3.2 Approach Difficulties

There are a number of difficulties to our approach of building structures which represent combinations of rule execution. For example, for a contract to match on a combination of rules, it may be necessary to unify the elements from different rules. That is, to prove the contract in Figure 5, it will necessary for a *Country* element from two rules to have executed over the same input element.

Another difficulty to be resolved is that a rule may be required to be applied multiple times on an input model to produce the correct number of output elements for the contract to hold. Our verification technique is able to represent multiple applications of rules (Section 7.3.2), if the number of applications is known ahead of the path condition construction process.

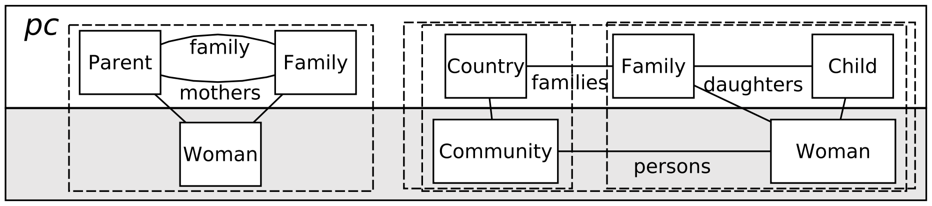


Figure 7: A path condition which is a counter-example to the contract in Figure 6b

3.3 Prover Implementation

Our research has lead to the development of a number of tools to prove structural contracts on DSLTrans transformations. Figure 8a and Figure 8b show two front-end plugins which are used to build DSLTrans transformations and contracts. Another option is to directly translate a transformation from the ATL language to DSLTrans using a higher-order transformation [46].

The first plugin (Figure 8a) is for the Eclipse platform [4], which allows our prover to be integrated with the rich Eclipse ecosystem. This furthers the applicability of our technique, as there are multiple model transformation tools such as ATL [2], EGL [5], and VIATRA [15] which are integrated with the Eclipse Modeling Framework (EMF) [3].

The plugin in Figure 8b shows the front-end for our contract prover in the Meta Programming System (MPS) editor [34]. MPS is designed for the creation of domain-specific languages, which can ease the development of software [57]. The DSLTrans and contract languages have been recreated within MPS to provide the necessary concepts. One interesting feature of MPS is that this explicit modelling of the languages provides a number of desirable features, such as auto-completion and type checking in the projectional editor. Note that we found that creating the transformation and contracts through the projectional editor is significantly faster and less error-prone than other creation methods.

These frontends have been integrated with a backend which implements the principles laid out in this paper. The transformation and contracts are transferred to the prover (currently implemented in Python), which generates the path conditions and performs contract proof. The counter-examples to the contracts are then sent back to the front-end.

We refer the reader to our past work for more information on the implementation and architecture of our technique, including performance results on a number of transformations [41, 46]. These results indicate that our technique is scalable and widely applicable, as we can prove multiple contracts on relatively large transformations within the span of a few minutes.

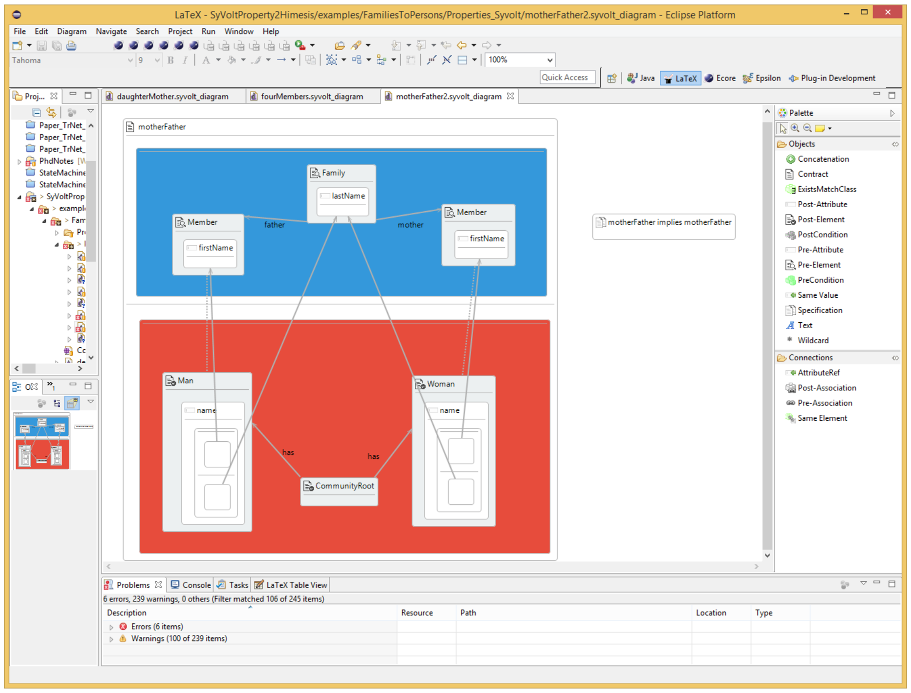
In particular, we have previously explored the technique of 'slicing' the transformation based on the contract to be proved [46]. This approach is a static analysis to determine which rules need to execute (and how many times) for successful contract validation. As seen in that work, this optimization can dramatically reduce the time required to prove contracts.

4 DSLTrans Formalization

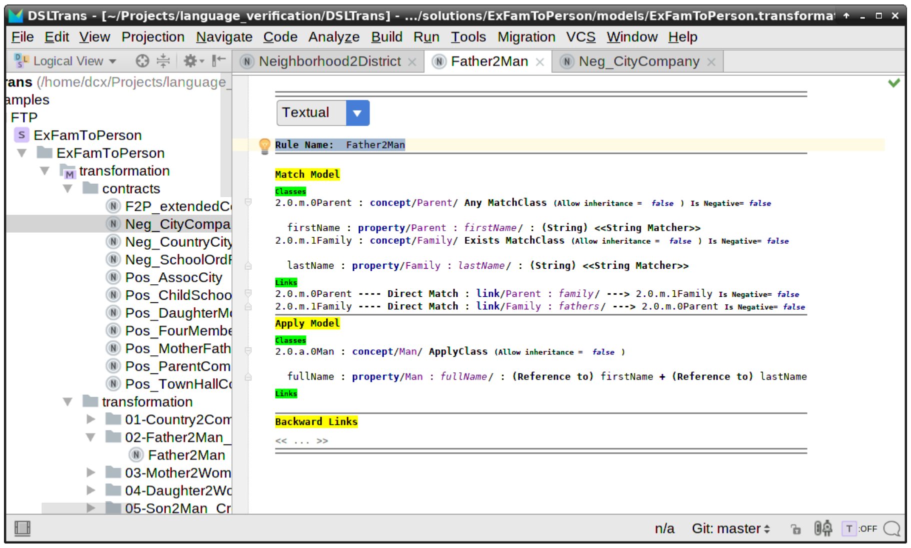
This section reproduces (from Oakes *et al.* [45]) the formal background and structures of DSLTrans. These structures are necessary to explain the symbolic-execution process which we use to verify DSLTrans transformations.

We begin our formalization by introducing (typed) graph concepts that will be used as mathematical building blocks. In particular we introduce our representation of typed graphs and the homomorphisms we use between them, based on well-known concepts from graph theory, such as in Ehrig *et al.* [24].

Armed with these fundamental definitions, we then introduce the formal concepts of the DSLTrans transformation language in Section 5.



(a) Eclipse frontend



```
Rule Name:  Father2Man

Match Model
Classes
2.0.m.0Parent : concept/Parent/ Any MatchClass (Allow inheritance =  false ) Is Negative= false

  firstName : property/Parent : firstName/ : (String) <<String Matcher>>
2.0.m.1Family : concept/Family/ Exists MatchClass (Allow inheritance =  false ) Is Negative= false

  lastName : property/Family : lastName/ : (String) <<String Matcher>>
Links
2.0.m.0Parent ---- Direct Match : link/Parent : family/ ---> 2.0.m.1Family Is Negative= false
2.0.m.1Family ---- Direct Match : link/Family : fathers/ ---> 2.0.m.0Parent Is Negative= false
Apply Model
Classes
2.0.a.0Man : concept/Man/ ApplyClass (Allow inheritance =  false )

  fullName : property/Man : fullName/ : (Reference to) firstName + (Reference to) lastName
Links

Backward Links
<< ... >>
```

(b) MPS frontend

Figure 8: Frontends for our prover tool

We also note that we require the formalization of DSLTrans to properly explain how we can prove contracts over transformations. In particular, we rely on the description of the semantics of DSLTrans to justify our symbolic representation of rule execution (Section 7). Bentley ▶***Keep?***◀

4.1 Typed Graphs

Typed Graph

The typed graph is the essential object we will use throughout our mathematical development to formalize all the important graph-like structures, such as rules and input-output models. A typed graph is a directed multigraph (a graph allowing multiple edges between two vertices) where vertices and edges are typed.

Definition 1. *Typed Graph*
A typed graph is a 6-tuple
$\langle V, E, (s,t), \tau, VT, ET \rangle$ *where:*

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

4.1.1 Vertex and Edge Types

The DSLTrans transformation language is targeted at transforming domain-specific models in the model-driven engineering domain. Therefore, the types of vertices and edges in our typed graphs will be drawn from a particular *metamodel*. The source and target metamodels for our example transformation are presented in Section 2.

These metamodels will play a part in Definition 7, which specifies how DSLTrans rules consist of multiple typed graphs with potentially different metamodels for the match and apply components.

For example, the match graph for the rule in Figure 4 is typed by the *Families_Extended* metamodel. Therefore, the elements and associations may have types such as *Member*, *Family*, and *fathers*, as seen in Figure 1b.

For the purposes of explaining rule semantics, we assume that the metamodel for a typed graph (such as the match or apply component) provides the necessary vertex and edge types, and that all duplicate types have been resolved through renaming. As well, we assume that the metamodel can be queried to provide a partial ordering $\leq$ on the vertex and edge types for a typed graph. This information will be critical for determining subtyping during graph matching. For example, both *Parent* and *Child* elements will be matched over by *Member* elements due to the inheritance relationship seen in Figure 1a.

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 (as in the RAMification procedure [37]) to perform our graph rewriting, as incomplete output graphs will be produced during the execution of the transformation.

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

- $\mathit{matchesOverType} : \{VT \cup ET\} \times \{VT \cup ET\} \rightarrow \{\mathit{true}, \mathit{false}\}$
  - The purpose of this function is to abstract the handling of polymorphism in the metamodel (defined by the partial ordering $\leq$)
  - $\mathit{matchesOverType}(t, t) \rightarrow \mathit{true}$
  - $\mathit{matchesOverType}(t, t') \rightarrow \mathit{true}$ iff t' is a subclass of t in the partial ordering $\leq$
  - Otherwise, *false*
- $\mathit{isIndirect} : ET \rightarrow \{\mathit{true}, \mathit{false}\}$
  - Where $\mathit{isIndirect}(et)$ returns *true* iff et is marked as an *indirect edge*, else *false*. The *indirect* classification allows for pattern-matching over indirect paths between vertices.
- $\mathit{isNegativeVertex} : V \rightarrow \{\mathit{true}, \mathit{false}\}$
  - Where $\mathit{isNegativeVertex}(v)$ returns *true* iff v in the graph is marked as 'negative'. This typing is used to create a *negative application condition* to prohibit the execution of DSLTrans rules (Section **??**).
- $\mathit{isNegativeEdge} : E \rightarrow \{\mathit{true}, \mathit{false}\}$
  - Where $\mathit{isNegativeEdge}(e)$ returns *true* iff a particular edge in the graph is denoted as 'negative'. This is used for negative match edges and negative backward links in DSLTrans rules.

Typed Subgraph

A typed subgraph is simply a restriction of a typed graph to some of its vertices and edges. This definition allows us to talk about partitioning DSLTrans constructs such as rules into logical components. For example, this definition can be used to select the match or apply components of a rule (Definition 7).

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* $g, g' \in \mathrm{TG}$*.*

g' is a typed subgraph of g, written $g' \sqsubseteq g$, iff $V' \subseteq V$, $E' \subseteq E$, $\tau' = \tau|_{V' \cup E'}$, $s' = s|_{E'}$, and $t' = t|_{E'}$.

Typed Graph Union

A union of two typed graphs is trivially the set union of all the components of those two typed graphs.

Definition 3. *Typed Graph Union*

The typed graph union is the function $\sqcup : \mathrm{TG} \times \mathrm{TG} \to \mathrm{TG}$ *defined as:*
$\big\langle V, E, (s,t), \tau, VT, ET \big\rangle \sqcup$
$\big\langle V', E', (s',t'), \tau', VT', ET' \big\rangle =$
$\big\langle V \cup V', E \cup E', (s \cup s', t \cup t'),$
$\tau \cup \tau', VT \cup VT', ET \cup ET' \big\rangle$

Note that $\cup$ is the normal set union, and $s \cup s'$, $t \cup t'$, and $\tau \cup \tau'$ must coincide on common elements. That is, if the same edge e appears in both E and E', then $s(e) = s'(e)$ and $t(e) = t'(e)$.

4.2 Graph Morphisms

The formal development of our technique requires the definition of relations between typed graphs that preserve (some) graph structure and vertex/edge type, i.e. morphisms.

As terminology, these morphisms find mappings or *matches* from vertices (and edges) in the *pattern graph* onto vertices (and edges) in the *target graph*.

Typed Graph Morphism

The morphism we present here is standard in the literature (such as [21]), where all elements of the pattern graph must be found in the target graph.

The first typed graph morphism we define is standard in the graph-matching literature, where the structure of the pattern graph is found isomorphically in the target graph [21].This homomorphism is presented for the reader to appreciate the complications required for the next morphism, which is not a standard isomorphism.

Definition 4. *Typed Graph Morphism*
Let $\langle V, E, (s,t), \tau, VT, ET \rangle = g$, *and*
$\langle V', E', (s',t'), \tau', VT', ET' \rangle = g'$, *where* $g, g' \in \mathrm{TG}$.

A typed graph morphism $g \blacktriangleright g'$ *is a function* $f : f_v \cup f_e$ *such that:*

- $f_v : V \to V'$;
- $f_e : E \to E'$;
- $\forall v_1 \xrightarrow{e} v_2 \in E$:
  - *Let* $v_1' = f_v(v_1), v_2' = f_v(v_2), e' = f_e(e)$;
  - $v_1', v_2' \in V', e' \in E'$;

- $s'(e') = v'_1$, $t'(e') = v'_2$;
- $matchesOverType(\tau(v_1), \tau'(v'_1)) \wedge$
  $matchesOverType(\tau(v_2), \tau'(v'_2)) \wedge$
  $matchesOverType(\tau(e), \tau'(e'))$.

This last condition handles matching of types, such that the vertex and edge types in the pattern are the same or superclasses of the types in the target graph. That is, the types of the vertices (or edges) are obtained with τ or τ'. Then, the matchesOverType function responds whether the types match.

An *injective*[1] typed graph morphism from g onto g' is written $g \overset{inj}{\blacktriangleright} g'$. A *surjective* typed graph morphism from g onto g' is written $g \overset{surj}{\blacktriangleright} g'$.

Typed Graph Split Morphism

The improvements in this article to the path condition creation technique requires a form of graph morphism that primarily focuses on matching the edges of the graphs. This morphism will still 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. Thus, this morphism performs 'one-to-many' matching.

A concrete example for when this morphism is required is found in Section 9.2.3. Briefly, the issue is that a contract needs to be matched onto a structure representing multiple rules. Therefore, one element in the contract may match onto multiple elements in the rules. This is our way of unifying elements, without the combinatorial explosion of generating all unifications of the rules explicitly as is performed by Barroca [12].

Definition 5. *Typed Graph Split Morphism*
Let $\langle V, E, (s,t), \tau, VT, ET\rangle = g$, *and*
$\langle V', E', (s',t'), \tau', VT', ET'\rangle = g'$, *where* $g, g' \in \mathrm{TG}$.

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

- $h_v : V' \to V$ *- a function with 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 match onto vertices in the pattern graph*
  - *Surjective so that all pattern vertices are matched to by target vertices*
- $h_e : E \to E'$ *where h_e is injective*
- $\forall v_1 \xrightarrow{e} v_2 \in E$:
  - *Let* $e' = h_e(e), v'_1 = s'(e'), v'_2 = t'(e')$

[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.

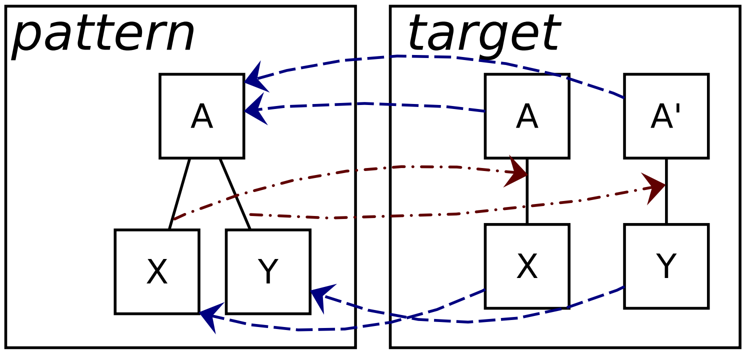


Figure 9: An example of the typed graph split morphism.

– $matchesOverType(\tau(v_1), \tau'(v'_1)) \wedge$
  $matchesOverType(\tau(v_2), \tau'(v'_2)) \wedge$
  $matchesOverType(\tau(e), \tau'(e'))$

When a typed graph split morphism f exists from g onto g', we write $g \rhd g'$.

Again, we wish to highlight the fact that this morphism 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.

An example of this morphism is seen in Figure 9. The left-to-right arrows indicate edge matching, while the right-to-left arrows demonstrate how multiple target elements match onto the same pattern element.

This unusual morphism 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 represents this vertex duplication. That is, two vertices of a particular type in the target graph may only represent one underlying element.

The motivation for requiring this 'one-to-many' matching is provided in Section 6. As a summary, we require this morphism to avoid producing all possible rule overlaps during path condition construction.

4.2.1 Link Morphism

For convenience, we also define a function to match components containing *links*.

In the structures we use in DSLTrans, these links connect elements in the output graph with elements in the input graph. An example would be the backward link construct for DSLTrans rules, as discussed in Section 2.3. This morphism is necessary such that the backward links from Definition 7 can match onto the component containing traceability links from Definition 10.

Definition 6. *Link Homomorphism*
Consider two tuples of links $A = \{E_A, (s_A, t_A)\}$, and $B = \{E_B, (s_B, t_B)\}$.

We define a (trivial) injective homomorphism function f to match A over B, such that:

- $\forall e_a \in E_A$:
  - $\exists e_b \in E_B \mid (f(e_a) = e_b)$ *and that the constraints (types, attributes) match for the source and target nodes of the links.*

4.2.2 Morphisms Between Structures

In the developments that follow, we will need to find morphisms between different structures. For example, these structures may be a DSLTrans rule (Definition 7) or an input-output model (Definition 10).

We create a morphism m between two structures $S_1 = \langle A, B, L\rangle$ and $S_2 = \langle A', B', L'\rangle$, where $A, A', B, B' \in TG$ and L, L' are links of the form $\{E, (s, t)\}$.

This overall morphism is composed of the sub-morphisms $A \xrightarrow{f} A'$, $B \xrightarrow{g} B'$, $L \xrightarrow{h} L'$, where f and g are the morphisms to be found, and h is the link homomorphism described in Definition 6. Note that f and g may have differing properties, such as one being injective or not. Therefore, we consider it more elegant to require three distinct sub-morphisms. Our definitions for these structures require that the components (such as A, B, and L) are all disjoint. Thus we consider the morphism to be trivially composed: $m = f \cup g \cup h$.

However, we require these morphisms to have consistency in their vertex mapping for m to be valid.

For example, let link $l \in L$ be the pattern link, with source and target vertices $s(l) \in A$ and $t(l) \in B$. As well, let there be a target link $l' \in L'$ where $h(l) = l'$, with source and target vertices $s'(l') \in A'$ and $t'(l') \in B'$. Then, the composed morphism will be valid iff $g(s(l)) = s'(l')$ and $f(t(l)) = t'(l')$.

That is, if a link is matched by the link morphism h, then the correct source and target vertices must be matched by the f and g morphisms.

5 DSLTrans Structures

This section will detail the constructs involved in a DSLTrans transformation such as rules and layers.

5.1 Similar Structures

Note that in following sections, we define structures which are very similar in composition, with each containing two typed graphs, as well as a link component. This similarity is essential to our proving technique as we will define morphisms between the various components of these constructions.

This approach of typed graph composition is preferred by the authors due to its reflection in our implementation. Note that it is equivalent to think of these structures as one large typed graph.

In this case, elements would be annotated by the component they originate from, and Definition 2 (Typed Subgraph) can be used to select an appropriate projection.

DSLTrans Transformation Rule

Recall from Section 2 that DSLTrans transformations are composed of rules arranged in layers.

A transformation rule includes a match graph and a non-empty apply graph, as seen in an example rule in Figure 2. These match and apply graphs are also known in the model transformation literature as a rule's *left-hand side* and *right-hand side*. As described in Section 2, when the elements from the rule's match graph are found in the input model, then the apply graph elements are produced in the output model.

Match graphs can also include indirect links that are used to transitively match associations in a model, or a rule may contain negative elements, such as match elements, direct and indirect match links, or backward links.

An apply graph does not include indirect links as it is used only for the construction of parts of instances of a metamodel.

A transformation rule also includes backward links (and potentially negative backward links) to define dependencies between rules, as introduced in Section 2.3. These links will be placed in a separate component in the formalized DSLTrans rule.

Definition 7. *DSLTrans Transformation Rule*
A DSLTrans transformation rule is a three-tuple $\langle Match,\ Apply,\ backward\rangle$, where:

- $Match, Apply \in \mathrm{TG}$*;*
- *Match and Apply are disjoint, and Apply is non-empty.*

Note that when we require an element of the Match or Apply 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}*.*
- (s_{back}, t_{back}) *is a pair of functions*
  $s_{back} : E_{back} \to V_{Apply}$ *and* $t_{back} : E_{back} \to V_{Match}$ *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.*

DSLTrans Layer and Transformation

Definition 8 and Definition 9 formalise a DSLTrans transformation, which is composed of a sequence of layers where each layer is composed of a set of rules.

Definition 8. *Layer*
A layer is a finite set of transformation rules: $\{r_0, r_1, \ldots, r_n\} \mid r_i \in \textsc{Rules}$.

The set of all layers is denoted $\textsc{Layers}$.

Note that the order of the rules within the layer does not matter, due to the semantics of DSLTrans rule execution.

Definition 9. *DSLTrans Transformation*
A DSLTrans transformation is a finite list of layers denoted $(l_0, l_1, \ldots, l_n) \mid l_i \in \textsc{Layers}$. *Note that the order of layers in a transformation is important, as this ordering defines the execution of the transformation.*

A transformation is also associated with an input metamodel MM_{in} *and an output metamodel* MM_{out} *to provide typing information for the input and output models.*

The set of all transformations is denoted as $\textsc{Transforms}$.

Input-Output Model

Before describing the semantics of a DSLTrans model transformation in Section **??**, we must first define the central *input-output model* construct. This construct represents the input model given as input to the transformation, as well as the intermediate operational states during the transformation's execution.

The structure of a input-output model is intentionally very similar to a DSLTrans rule to support the matching and rewriting of elements, as discussed in Section **??**. An input-output model contains 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.* These links indicate which elements in the output model originated from which elements in the input model.

During execution of the transformation, a rule will be matched onto the input-output model to determine if the rule can execute. If so, the appropriate output elements will be created in the output portion. As well, traceability links will be created between the output elements and those input elements involved in the rule. These semantics are described in Section **??**.

As DSLTrans rules cannot delete elements or links, an input-output model can only accumulate output elements and traceability links as the transformation proceeds.

Definition 10. *Input-Output Model*
An input-output model rule is a three-tuple $\langle Input,\ Output,\ trace \rangle$*, where:*

- $Input, Output \in \mathrm{TG}$*;*
- *Input and Output may be empty and are disjoint;*
- $trace = \{E_{trace}, (s_{trace}, t_{trace})\}$
  - E_{trace} *contains the traceability links;*

– *E_{trace} is disjoint from E_{Input} and E_{Output};*

– *(s_{trace}, t_{trace}) is a pair of functions*
$s_{trace} : E_{trace} \rightarrow V_{Output}$ and $t_{trace} : E_{trace} \rightarrow V_{Input}$ that respectively provide the source and target vertices for each traceability link.

Note that as this structure represents an execution of the transformation, the vertex and edge types VT and ET for the input and output graphs will be sourced from the input and output metamodels for the transformation.

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

We define a utility function getTransformation: IOM $\rightarrow$ Transforms. This (trivial) function returns the transformation that the input-output model was built for. The purpose of this function is to restrict input-output models to be applicable for only the transformation they represent, as the output model was created through the execution of rules in that transformation.

6 Abstraction Relation

This section will present our technique for representing the infinite number of possible executions of a DSLTrans transformation. That is, there are an infinite number of possible input-output model pairs, each of which is finite.

Our technique creates a finite set of representations to represent these input-output models. These representations are termed *path conditions*, and they are structured to symbolically represent the applications of rules.

As well, this section defines the abstraction relation between a path condition and the set of transformations executions that it represents. This abstraction relation allows our technique to prove contracts on the finite set of path conditions created by the path condition generation algorithm. As this set is finite, our technique is guaranteed to take a finite amount of time. Our procedure for proving contracts is discussed in Section 9.

6.1 Symbolic Execution

Our algorithm operates on the principle of symbolic execution to build the set of path conditions for the transformation, in an analogy with program symbolic execution. As introduced by King in his seminal work "*Symbolic Execution and Program Testing*" [35], 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 transpose this notion of symbolic execution to model transformations. The analog of an input variable in the model transformation context are *metamodel classes*, *associations* and *attributes*. As program statements impose constraints on input and output variables during symbolic execution, transformation rules impose conditions on which metamodel elements are instantiated during a concrete transformation execution, and how that instantiation happens. As well, the dependencies present in rules 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 6.4.

6.2 Rule Combinations

To present the intuition of path conditions and symbolic executions, we first discuss the idea of *rule combinations*. This is to present the idea of explicitly representing each application of the rule (one match-rewrite production), and recording the input and output elements induced by rule application.

As described in Section 2, a layer in a DSLTrans transformation contains a set of rules. We can create a set of rule combinations for this layer, where each rule combination in this set will represent all possible transformation executions where the rules in that combination would execute.

For example, in Figure 10, the rule combination marked AB represents the set of transformation executions with at least one application of rules A and B. Another rule combination marked A represents the transformation executions where only rule A has applied at least once. Another rule combination is marked AA representing at least two applications of the A rule.

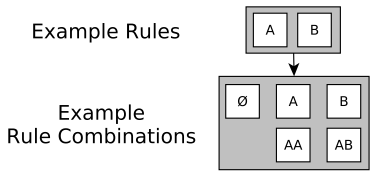


Figure 10: Rule combinations created for a transformation layer

Note that within these rule combinations, the number of times a rule has applied is necessarily abstracted. This is because we consider all possible input models, and therefore it is impossible to create combinations for all possible rule applications. In our technique, the abstraction will be a lower bound on the number of times a rule applies.

In other words, if the rule is not represented in a rule combination, this represents the case where the rule does not apply on the input model. If the rule is present once, then this represents the executions where the rule applies once or more in the input model. Likewise, a rule which appears multiple times in a rule combination represents the case where it has applied at least that many times in the input model.

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. Briefly, the path conditions will provide the input and output elements induced by rule application, as demonstrated in Section 6.3.

We also note that we are not concerned with representing the order of rule application within a rule combination. This is due to the confluence property of DSLTrans described in Section 2, which allows us to ignore rule ordering within a layer.

We base our concept of path conditions on these rule combinations. However, as DSLTrans allows for dependencies between rules, we cannot create path conditions by creating rule combinations for all rules in the transformation. Instead, our approach must move layer-by-layer and resolve the dependencies between rules.

6.3 Path Conditions

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 application 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 applied, or has applied at least some number of times.

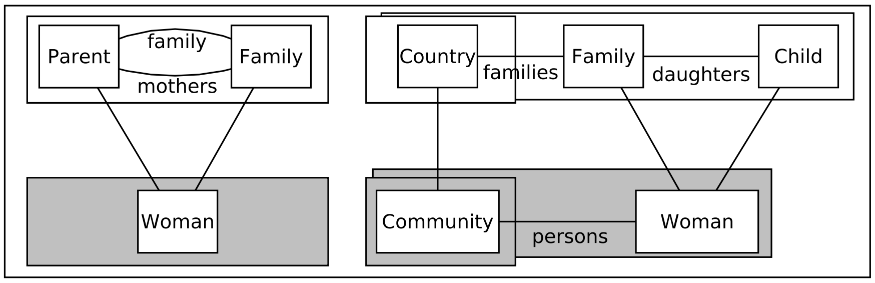


Figure 11: An example path condition representing the symbolic execution of three rules (*MotherToWoman*, *CountryToCommunity*, *DaughterToWoman*).

For example, the path condition in Figure 11 represents the execution of three rules in the *Families-to-Persons* transformation: *MotherToWoman*, *CountryToCommunity*, and *DaughterToWoman*. This representation includes the input and output elements that will be present in the input and output models if these three rules execute.

Note that we are representing a lower bound on the number of times a rule has applied on an input model. This means that this path condition also represents those transformation executions where the three rules have applied any number of times.

A formal definition of path conditions is presented in Definition 11. Following this, the abstraction relation is defined in Definition 12, and multiple examples of path conditions representing transformation executions are found in Section 6.5.

Definition 11. *Path Condition*

A path condition is a three-tuple $\langle Input,\ Output,\ trace \rangle$*, where:*

- $Input, Output \in \mathrm{TG}$
- *Input and Output may be empty and are disjoint*
- $trace = \{E_{trace}, (s_{trace}, t_{trace})\}$

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

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 for the path condition to hold. Note that similar to DSLTrans rules, a path condition may contain negative elements in the input graph. This is because the path condition will be composed of rule components.

The *Output* graph of a path condition is a pattern which will be instantiated in the output model of the transformation if the *Input* graph is present in the input model.

Symbolic traceability links are also kept between elements in the *Input* and *Output* graphs to retain traceability information. This replicates the traceability information found in DSLTrans rules as discussed in Section 2.

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

The set of all path conditions is noted as PATHCONDS.

We define a utility function GETRULE: $V \rightarrow$ *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.

As well, this function allows the generation of the rule set represented by the path condition. This information will be used to report the status of contract proof.

We also define a utility function GETTRANSFORMATION: *PathCond* $\rightarrow$ *Transforms*. This function returns the transformation that the path condition was built for. The purpose of this function is to restrict path conditions to only be applicable for the transformation they represent.

6.4 Abstraction Relation

The abstraction relation is at the core of our symbolic execution technique. It allows us to represent an infinite set of transformation executions with a finite set of path conditions. 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 9.

As can be seen in Figure 11, path conditions represent a set of transformation executions by storing the elements that will be present in the input and output models for that execution, along with traceability links which store which rules have executed.

The abstraction relation will map:

- all input elements and traceability links in the path condition to the input model of the transformation execution;
- all output elements and traceability links in the transformation execution to the output model of the path condition.

The abstraction relation partitions the infinite set of transformation executions into separate path conditions based on which rules have symbolically executed in the path condition, and therefore which elements will be present in the input and output models of those transformation executions.

However, these partitions may overlap, as path conditions represent a lower bound on the number of times a rule applies. For example, one path condition would abstract all transformation executions where rule A applies at least once, while another path condition would abstract all transformation executions where rule A applies two or more times. Therefore there is an ordering to path conditions based on the abstraction of the multiplicity for rule application.

To put this another way, a transformation execution may be abstracted by multiple path conditions. In Section 9 we will discuss how this affects the contract verification process, which relies on the abstraction relation to extend the proof result on one path condition to the abstracted transformation executions.

6.4.1 Formal Definition

This section will formally define the notion of abstraction of an input-output model by a path condition. Section 6.5 will then present examples for each condition in the definition.

The abstraction relation has two main stages. The first stage is to ensure that all input elements in the path condition can be found in the input-output model. The second stage checks to ensure that all output elements and all rules in the input-output model are represented by the path condition.

Definition 12. *Abstraction of a Input-Output Model by a Path Condition*

Let $iom = \langle Input,\ Output,\ trace\rangle \in \mathrm{IOM}$ *be an input-output model.*

Let also $pc = \langle Input,\ Output,\ trace\rangle \in \textsc{PathConds}$ *be a path condition.*

Trivially, we also require that getTransformation*(iom) and* getTransformation*(pc) return the same transformation.*

We have that iom is abstracted by pc, noted $pc \mapsto iom$*, if and only if the following conditions are all true:*

Condition 1

pc_{Input} matches onto iom_{Input}.

The purpose of the relation defined in Equation 1 is to enforce that the path condition accurately represents the elements present in the input model of the input-output model. As well, note that the negative elements present in pc_{Input} must not match.

$$\begin{aligned} \exists f : (pc_{Input} \underset{f}{\blacktriangleright} iom_{Input})\ where \\ \forall v_1, v_2 \in V_{pc_{Input}} : \\ f(v_1) = f(v_2) \implies getRule(v_1) \neq getRule(v_2) \end{aligned} \quad (1)$$

Extra elements may be present in the input-output model but not matched. These extra elements represent elements that will not be matched by any executing rules.

An important consideration for the abstraction relation is that the path condition is constructed through a composition of rule elements. Note that the second and third lines of Equation 1 restrict the matching of vertices from the same rule. If vertices overlap then they cannot belong to the same rule in the input, as given by *getRule*. However, elements may overlap between rule components. Rule components are presented as elements bounded by dashed lines in the figures in Section 6.5.

Condition 2

Elements in iom_{Output} must surjectively match on the pc_{Output}.

$$\exists g : pc_{Output} \overset{surj}{\underset{g}{\blacktriangleleft}} iom_{Output} \tag{2}$$

This surjection relation g enforces that all elements which have been created in the output of the input-output model must be represented by the output of the path condition. The surjective nature of the relation also means that all elements of the path condition must be matched onto.

There may be multiple copies of an element in the input-output model, but the path condition may represent the application of rules fewer times. In this case, these elements in the input-output model may match over the same element in the path condition. This multiplicity abstraction is discussed in Section 6.6.2.

Condition 3 Traceability links in the path condition must injectively match onto links in the input-output model using the match function t defined in Equation 3. The purpose of this condition is to ensure that the rule execution represented in the path condition reflects the actual rule execution performed in the input-output model.

$$\begin{aligned} \exists t : pc_{trace} \overset{link}{\underset{t}{\blacktriangleright}} iom_{trace}\ where \\ \forall e_1, e_2 \in pc_{trace} : \\ t(e_1) = t(e_2) \implies getRule(e_1) \neq getRule(e_2) \end{aligned} \tag{3}$$

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

Condition 4 Traceability links in the input-output model must surjectively match onto links in the path condition with the homomorphism u defined in Equation 4. This condition is to ensure that there is no rule execution in the input-output model which is not represented by the path condition.

$$\exists u : pc_{trace} \overset{link}{\underset{u}{\blacktriangleleft}} iom_{trace} \tag{4}$$

Condition 5 The morphisms used in the above conditions must be consistent with each other and agree over common elements.

$$f \uplus g \uplus t \uplus u \tag{5}$$

6.4.2 Partitioning of Executions

The abstraction relation therefore partitions the infinite set of transformation executions into separate path conditions based on which rules have symbolically executed in the path condition, and therefore which elements will be present in the input and output models of those transformation executions.

However, these partitions may overlap, as path conditions represent a lower bound on the number of times a rule applies. For example, one path condition would abstract all transformation executions where rule A applies at least once, while another path condition would abstract all transformation executions where rule A applies two or more times. Therefore there is an ordering to path conditions based on the abstraction of the multiplicity for rule application.

To put this another way, a transformation execution may be abstracted by multiple path conditions. In Section 9 we will discuss how this affects the contract verification process, which relies on the abstraction relation to extend the proof result on one path condition to the abstracted transformation executions.

6.4.3 Other DSLTrans Constructs

We must also consider the presence of other DSLTrans constructs in the rules, namely attributes, and *Any*/*Exists* elements. We refer to Oakes *et al.* [45] for further discussion of these constructs.

DSLTrans rules may also include attributes on the match elements in rules. For elements in the path condition to abstract over the elements in the input-output model, we assume that there is a proper solver that can determine the status of this attribute matching.

The *Any*/*Exists* elements denotes different semantics for the matching of multiple executions of the same rule. If there are multiple copies of the same rule in the path condition, and there is an element marked as an *Exists* element, then all copies of that element must map to the same element in the transformation execution.

6.5 Examples

In this section, we provide a number of examples to demonstrate the workings of the abstraction relation. In each figure, the path condition *pc* is on the left-hand side, and an input-output model *iom* is on the right-hand side.

Figure 12 presents the legend for the homomorphisms in the following figures. These homomorphisms will show potential matches between elements, while an X placed over the homomorphism arrow means it cannot be satisfied. The presence of these matches allow the conditions of the abstraction relation to hold.

We also note that matches must also match over associations between the elements. However, this is not included in the figures to avoid further visual cluttering.

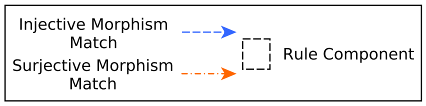
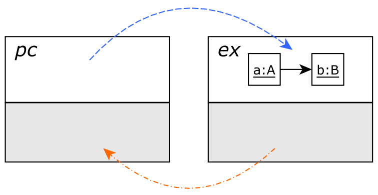


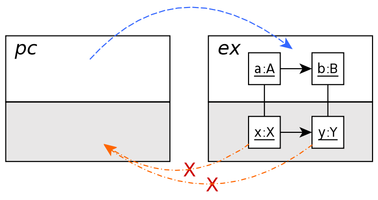
Figure 12: Legend for abstraction relation figures.

6.6 Example 1 – Empty Path Condition

The first example in Figure 13 demonstrates how an empty path condition will abstract over all input-output models where no rules execute. Note that in Figure 13a, the path condition abstracts the input-output model, while in Figure 13b, the abstraction relation does not hold.



(a) Abstraction holds



(b) Abstraction does not hold

Figure 13: Abstraction of input-output models by the empty path condition

The input part of the path condition represents the pre-conditions for the path condition to be true, depending on which rules have symbolically executed in the transformation. Since the input graph is empty, the empty path condition represents all input-output models 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 input graph of the path condition, and an input-output model. Note that in both Figure 13a and Figure 13b, an empty typed graph homomorphism satisfies this condition, denoted by the blue arrows.

The second condition for the abstraction relation is whether a typed graph surjective homomorphism can be found from the output model to the output graph of the path condition. This is represented by orange arrows in Figure 13a and Figure 13b. This relation is surjective as there may not be any elements in the output model that are not represented by the path condition's output graph.

The empty output 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 in Figure 13a 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.

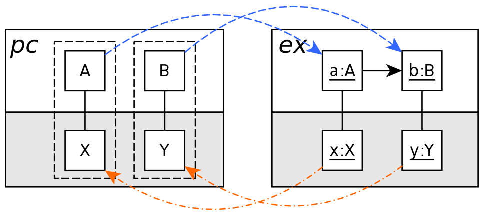
In contrast, in Figure 13b no surjection homomorphism can be found between the elements of the output model and the path condition. As the input-output model has elements in the output model, at least one rule must have executed. However, the path condition does not represent that a rule has executed, and therefore cannot abstract over this input-output model.

Note that this empty path condition is of great interest to our contract proving approach. It represents all input-output model pairs where the input model is insufficient for any transformation rules to execute. All input-output models not abstracted by another path condition will necessarily be represented by the empty path condition, as described in Section 8.2.

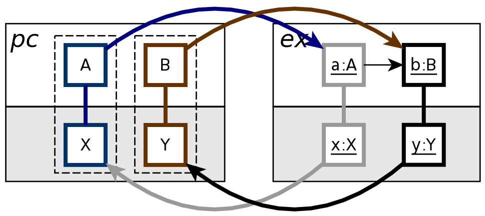
6.6.1 Example 2 – Non-overlapping Rule Components

This second example shows the abstraction relation between path conditions and input-output models with the presence of elements in the path condition. However, we restrict the elements to be distinct, such that no match element of the same type appears in multiple rule components.

Note that to properly represent the four conditions of the abstraction relation, two figures are displayed for the following examples. The first will demonstrate the matching performed on the input and output graphs, while the second figure focuses on the matching of the traceability links.



(a) Abstraction holds on non-overlapping elements.



(b) Abstraction holds on traceability links.

Figure 14: Abstraction of input-output models by non-overlapping rule components

Let us first examine how the morphism operates between the input elements in the path condition and the input-output models in Figure 14a. Note that this morphism can be found from the elements successfully. Similarly, there is a surjection morphism that can be found between the elements of the output graph and the output graph of the path condition.

We now examine Figure 14b to resolve whether the traceability links in the path condition can be found in the input-output model. 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 input-output model.

As well, there is a matching step from each individual traceability link in the input-output model onto the path condition. Similar to the matching from the path condition, the components in the input-output model figure are successfully matched onto the path condition.

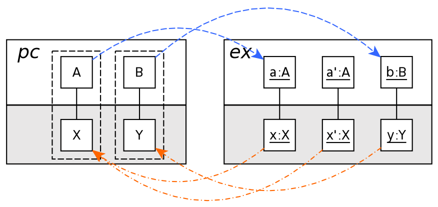
6.6.2 Example 3 – Multiple Rule Execution

Multiplicities must be correctly handled in our approach, such that a path condition can abstract over input-output models where a rule applies multiple times.

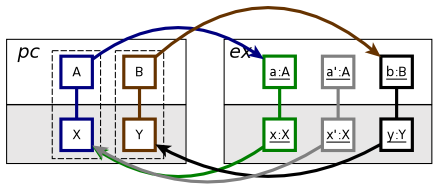
Recall that rule application is represented as either not occurring, or as a lower bound in the path condition. That is, if a rule does not appear in a path condition, the rule does not apply in the input-output model. If the rule appears once, it applies one or more times. If the rule appears twice, it applies two or more times, and so on.

Figure 15a and Figure 15b show an example of the abstraction relation where the rule applies multiple times in the input-output model. Note that multiple copies of the rule in the path condition can be easily matched to the input-output model in Figure 15a, as long as the rule multiplicity in the path condition is a lower bound for the input-output model.

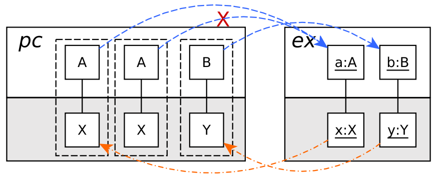
Note that this surjective match also holds in Figure 15b, where examination of the input-output model shows that one rule has applied twice. As mentioned before, the abstraction relation abstracts over the number of times that a rule has applied, such that multiple elements in the execution's output model may match to the same element in the output graph of the path condition. This is expected, as the structure found in the path condition's output graph may be found multiple times in the output model.



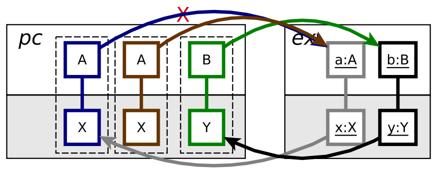
(a) Representing a lower bound of rule application.



(b) Traceability links can be found.



(c) Cannot represent more rule applications then have occurred.



(d) The traceability links also cannot be matched.

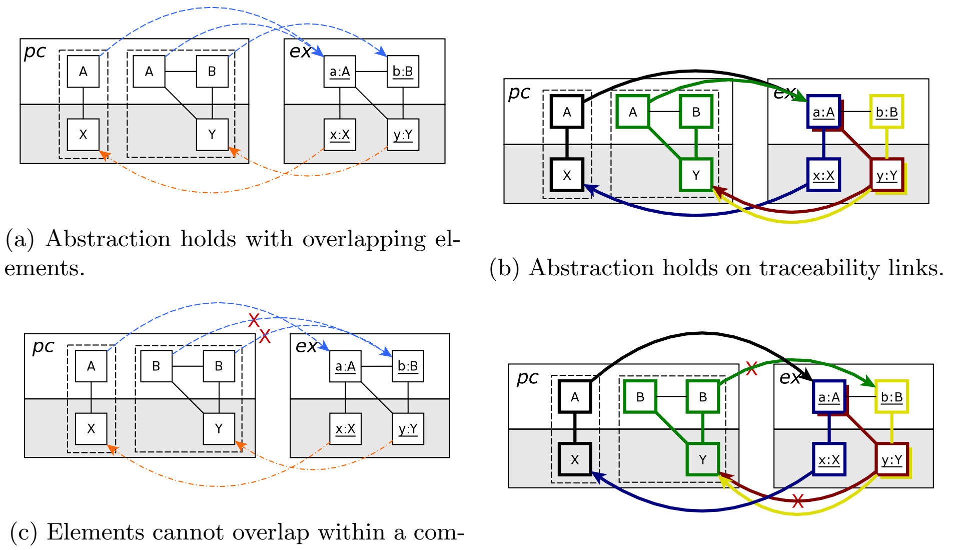
Figure 15: Example of abstraction over multiple rule execution

In contrast, Figure 15c and Figure 15d demonstrates how the abstraction relation will not hold if the rule is symbolically executed fewer times in the path condition than the input-output model. This is because the elements from both the duplicated $A \rightarrow X$ rules cannot match over the same elements in the input-output model. Thus, a path condition which represents more rule applications than the input-output model cannot abstract over that input-output model.

6.6.3 Example 4 – Overlapping Rule Components

For this example, the path conditions contain rule components with elements of the same type. Our goal is to illustrate the interaction of rule elements, where the path condition elements may match over the same or different elements in the input-output model.

For example, the two rule components in Figure 16a correctly match over the input-output model shown. The abstraction relation holds due to the fact that, while input elements of the same component need to be found injectively in the input-output model, the injection constraint does not span multiple components. This allows the input elements from different rules to match to the same input graph element in the input-output model.



(a) Abstraction holds with overlapping elements.

(b) Abstraction holds on traceability links.

(c) Elements cannot overlap within a component.

(d) Traceability links cannot be found.

Figure 16: Abstraction of input-output model by overlapping rule components

As well, Figure 16b shows the mapping from the path condition to the input-output model. However, note that the pattern composed of the A, B, and Y elements, along with the traceability links, is to be matched together (using the typed graph split homomorphism). This is to ensure that the traceability links are found in the proper configuration in the input-output model.

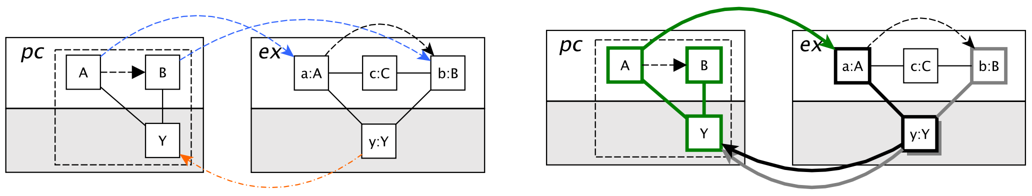
We also match the traceability links from the input-output model back onto the path condition. Again, this is to ensure that no traceability links are found in the input-output model that have not been represented in the path condition. Three matches are performed in this step, denoted by the three arrows in the bottom of Figure 16b. Each match is composed of a traceability link as well as immediately connected elements.

In contrast, Figure 16c shows an example where the abstraction relation does not hold. Note that one rule component in the input graph of the path condition contains two B elements. Both of these elements must be found in the input-output model, and thus it is not correct for them to match to the same element in the input-output model.

As well, it is informative to examine Figure 16d. Note the one of the matches from the input-output model 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 input-output model.

6.6.4 Example 5 – Indirect Links

Figure 17 presents a path condition that includes rules with indirect links. Note that the indirect link between elements $a : A$ and $b : B$ in the input-output model are added by the transitive closure we have defined in Oakes *et al.* [45]. This closure creates links between all nodes connected by a path, without loops. These links are shown as dashed lines between the elements in Figure 17.



(a) Matching on the link created. (b) Matching over traceability links.

Figure 17: Abstraction with indirect links

In this case, for the abstraction relation to hold, the elements at both ends of the indirect link must be found, and there must be a link created between the matched elements by the transitive closure that the indirect link can match on.

7 Building Path Conditions

Section 6 presents our abstraction relation such that once path conditions are built for a transformation, each path condition will abstract over an infinite set of transformation executions by defining which elements are present in those input-output models.

This section describes our symbolic execution approach for building these path conditions by examining interactions of rules, wherein all rules in the transformation are combined into path conditions by examining each layer in the transformation.

This abstraction of transformation executions by path conditions forms the basis for our technique of contract proving, where proving contracts on the set of produced path conditions allows us to reason about how the contract holds on the transformation's input-output model pairs.

Section 8 will then present arguments for the *coverage* and *representation* of the transformation executions, such that our contract proving approach is valid.

Rule Matcher and Rewriter Creating path conditions is based on the matching of rule *matchers*, and then the addition of the elements from the rule's *rewriter*. Note that for brevity, we refer the reader to Oakes *et al.* [45] for the formal definitions of a rule's matcher and rewriter.

The execution of a DSLTrans rule is based on the elements present in the match graph in the DSLTrans rule. As well, the backward links in the rule enforce dependencies on how the rule can match over the host graph. Therefore, the matcher for a rule r (denoted $\lceil r \rceil$) is composed of the rule's *Match* graph combined with backward links, as well as any *Apply* vertices connected to the backward links. Note that there are also negative elements and links in the match graph, as well as negative backward links. These elements will be present in the matcher, and will be matched during rule execution.

The construction of a rewriter for the rule r (denoted $\lfloor r \rfloor$) is essentially the same as the underlying rule. However we require that new traceability links be created between all match and apply vertices to indicate the proper traceability for the creation of elements. As well, we discard the negative elements in the rule, as they are used only for matching of the rule.

7.1 Path Condition Generation Algorithm

The aim of the path condition generation algorithm is to create a set of path conditions *PCSet* which symbolically represents the DSLTrans transformation under study. This set is built by iterating through each layer in the transformation, as outlined in Figure 18.

This algorithm will begin with *PCSet* as containing only the empty path condition. Then, the path conditions in *PCSet* will be combined with rules from the next layer to determine the symbolic execution of each rule. This combination process will produce a new set of path conditions for the layer *PCSet'*. Once all layers of the transformation have been examined, a complete set of path conditions for the entire transformation will have been produced.

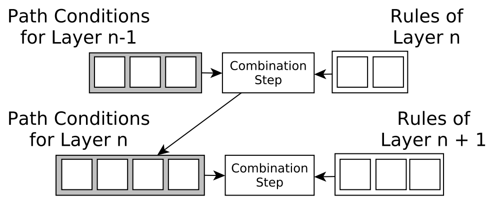


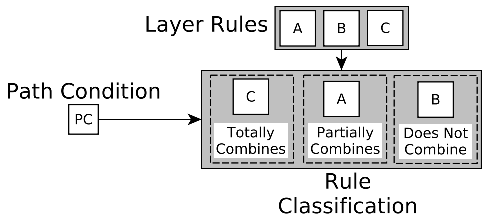
Figure 18: Previous path conditions are combined with rules

The *combination step* in Figure 18 is the core of our technique, where each path condition in *PCSet* is examined to determine how it combines with the current layer's rules.

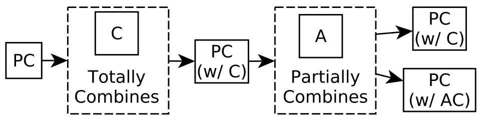
Our technique determines which rules are able to execute in the input-output models abstracted by that path condition. This is based on the input and output elements present in the path condition, which the rule may or may not apply over. If the rule can symbolically execute, then the input and output elements of the path condition are modified accordingly.

In an enhancement to our previous technical report [40], we present in this work a new and more intuitive approach to path condition construction. New path conditions will be created by defining sets of rules that the path condition *will* or *may* combine with. Then, these sets are used to create a new set of path conditions. Finally, we determine how the rules with negative elements combine with the path conditions.

An example is shown in Figure 19a, where the rules in a layer are classified as *totally combining*, *partially combining*, or *not combining* with the path condition. Figure 19b shows how path conditions from the previous layer are sequentially combined with all the rules from the current layer. Note that the *partially combining* set leads to an explosion in the number of path conditions.



(a) Classifying rules depending on the path condition.



(b) Creating new path conditions based on rule classification.

Figure 19: The path condition creation process for one path condition

7.2 Rule Combination Sets

We will now discuss the combination step, which partitions the rules in a layer into those that *may* combine with a particular path condition, and those that *will* combine.

Let *pc* be the path condition selected from layer $n-1$, and *rl* be a rule selected from layer n.

First, we add *rl* to the set $total_{pc}$ if *rl must* symbolically execute over *pc*. Similarly, *rl* will be added to the set $partial_{pc}$ if the rule *may* or *may not* execute over *pc*. These decisions are based on whether there is a morphism that can be found between the rule's matcher and the path condition.

Therefore, when *pc* and *rl* are combined, there are four possibilities based on the restrictions defined by the backward links and negative elements present within *rl*:

1. *rl* has **no** restrictions
2. *rl* has restrictions and **cannot** apply on *pc*
3. *rl* has restrictions and **may** apply on *pc*
4. *rl* has restrictions and **will** apply on *pc*

As mentioned in Section 2.3, backward links enforce that the elements matched by the rule's apply graph have been previously 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 element creation symbolically represented by the path condition.

The negative elements within the rule's match graph also define the conditions in which the rule cannot execute.

7.2.1 Total Combination

Definition 13 defines the $total_{pc}$ set, which holds all rules which will *totally combine* with the path condition *pc*.

Definition 13. *Path Condition and Rule Combination – Total Combination*

Given a path condition pc and a DSLTrans layer l containing rules:

$total_{pc} = \{rl \in l \mid (\exists back_{rl} \wedge \lceil rl \rceil \rhd pc \wedge \neg containsNegativeElements(rl)\}$

This definition selects those rules from the layer where 1) the rules specifies dependencies through backward links, and 2) the rule's matcher, including the links, totally matches over the path condition using the typed graph split morphism. Therefore, all elements required by the rule to execute are present in the path condition and thus in all abstracted transformation executions.

Note that if negative elements or links are present in the rule, then we cannot guarantee that the rule can match. This is because an input-output model abstracted by that path condition may contain extra input elements not captured in the path condition. Therefore, the rule is considered to may match, and so will be placed in the set $partial_{pc}$.

7.2.2 Partial Combination

Secondly, the $partial_{pc}$ set holds all rules which will *partially combine* with the path condition *pc*. The creation of this set is given by Definition 14.

Definition 14. *Path Condition and Rule Combination – Partial Combination*

Given a path condition pc and a DSLTrans layer l containing rules:

$partial_{pc} = \{rl \in l \mid \nexists back_{rl} \vee (\exists back_{rl} \wedge \lceil rl \rceil \not\rhd pc \wedge rl_{back} \overset{link}{\blacktriangleright} pc)\}$

This definition selects those rules where either the rules specifies no dependencies through its backward links, or there are dependencies specified, the matcher does not totally match over the path condition. As well in the second case, the backward links specified in the rule do match over the path condition, satisfying the rule's dependencies.

Therefore, the set $partial_{pc}$ *will contain rules where the input elements may be present in the transformation executions abstracted by the path condition.*

7.3 Rule Combination

Definitions 13 and 14 create the sets of rules to be combined with that path condition, defining whether the rule totally or partially combines with the path condition. Following this, the new set of path conditions for the layer can be created, which will then be the input to the combination step for the next layer.

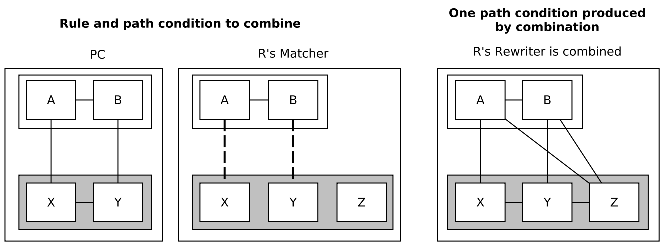
Note that there may be rules in the later that are in neither set. This is because rules may specify restrictions (using backward links) that cannot be satisfied by the path condition, and thus they will not be considered to combine with the path condition. This is the main way that the growth of the path condition set is restricted such that only feasible representations of the transformation are permitted.

7.3.1 Total Combination

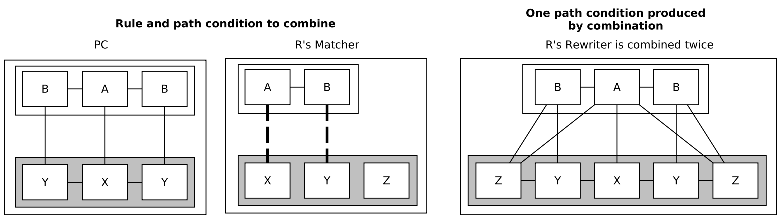
The total combination possibility represents the cases where the rule r totally matches over the path condition. Therefore, the elements required by the rule's matcher are present in the path condition, and the rule necessarily applies at least once.

The elements from the rewriter of the rule r will be added into the path condition pc, as seen in both subfigures in Figure 20. Note that this addition process is based on the double-pushout approach [24], such that new elements are created connected to where the rule's backward links overlap with the path condition's elements.

As well, multiple total matches of the rule on the path condition will cause multiple additions will occur as seen in Figure 20b.


Rule and path condition to combine
PC
R's Matcher
One path condition produced by combination
R's Rewriter is combined
A
B
X
Y
Z

(a) Totally satisfied at one location.


Rule and path condition to combine
PC
R's Matcher
One path condition produced by combination
R's Rewriter is combined twice
A
B
X
Y
Z

(b) Totally satisfied at multiple locations.

Figure 20: r's dependencies are totally satisfied by pc.

Definition 15. *Total Combination of a Path Condition and a Set of Rules*

The combination of a path condition pc and the rule set $total_{pc}$ *(defined by Definition 13) is described by the relation*
$PathCond \times \{Rule\} \overset{combine-total}{\rightarrow} \{PathCond\}$.

We create a mass production p^*_{total} from the rule set $total_{pc}$ following the definitions in Oakes et al. [45] and Ehrig et al. [24]. Note that it is permissible in DSLTrans to combine various rules together into one mass production as DSLTrans rules are confluent [11, 45]. In this case, the production p^*_{total} is composed of the matchers and rewriters of all the rules in the $total_{pc}$ set, with each rule represented as many times as it matches over the path condition pc.

This production is then applied to pc in Section 7.3.4, following the resolution of the partially-matching rules in $partial_{pc}$. The rule executes once in Figure 20a and twice in Figure 20b.

7.3.2 Partial Combination

Once all the rules which *must* execute on the path condition have been gathered into a path condition, the rules which *may* execute must be considered. This set of rules $partial_{pc}$ is slightly more complicated to handle, as there is a possibility that each of these rules does not execute in the abstracted transformation executions.

As seen in the total combination case, the objective is to generate mass productions which can be applied to the path condition *pc* to create appropriate rule elements and thereby symbolically execute rules. These mass productions are created out of the powerset of rule combinations from $partial_{pc}$.

Multiple Rules Before we can generate all the partial mass productions to be applied on the path condition, repeated rule application must be handled first.

In our technique, we are abstracting over the number of times that a rule applies to an input-output model. However, the contract we wish to prove may require particular rules to have applied more than once. Alternatively, the dependencies on a rule (the backward links) may require the application of another rule more than once.

In either case, this multiplicity information for rules must be decided before path condition construction begins. Then, the number of times a rule applies can be decided at this stage, during partial mass production creation.

We note that a relatively simple static analysis is sufficient to detect these situations and report which rules need to be duplicated. This analysis is implemented in our contract verification tool SyVOLT [41], where it is also used to detect invalid rules and contracts. Each contract and rule is decomposed to its constituent links and elements, and then searched for in other rules in the transformation. This allows us to determine which rules need to be duplicated (and how many times) to generate the appropriate number of elements for following rules to apply.

If rule application needs to be replicated, we add new rule combinations to the powerset which duplicate the rule in question. For example, if we want to duplicate the number of times that rule r applies to a path condition, we will duplicate all sets in the powerset with r, and add an extra (differentiated) copy of r within those sets. This ensures that we represent the multiple application of this rule.

Rule Subsumption If there are two rules on the same layer, and one subsumes the other, then they cannot both execute.

For example, consider rule A with a *Family* element and a *Member* element, and rule B with a *Family* element and two *Member* elements. It is clear that rule B can only execute on input models where rule A also executes. Therefore, we can remove rule combinations from the powerset where rule B executes but rule A does not.

Note that the determination of rule subsumption can be made before the path condition creation process in a static analysis.

Negative Combination Note that the set of rules $partial_{pc}$ that *may* combine with the path condition contains rules with negative elements. The presence of these negative elements indicates that there are input models with elements such that those rules cannot execute. As such, these rules cannot be placed in the $total_{pc}$ as they are not guaranteed to execute.

It would be preferable to remove as many invalid rule combinations as possible to create the minimum number of future path conditions. Therefore, we attempted to determine in what cases a rule with negative elements could not combine with other rules.

However, a guiding principle of our symbolic execution approach is that we are representing which elements in the input model will definitely exist when those rules execute. The abstraction relation (as discussed in Section 6) explicitly allows for the addition of extra elements in the input model which do not appear in the path condition. This means that it is incorrect to reason about which elements do not appear in an input model based on a path condition.

We present an example in Figure 21 of two rules that seem to be contradictory. The first rule is a *Country* element connected to a *Townhall* element, which produces a *ManagedCommunity* element in the output graph. The second rule is a *Country* element connected by a negative link to a negative *TownHall* element, which produces an *UnmanagedCommunity*.

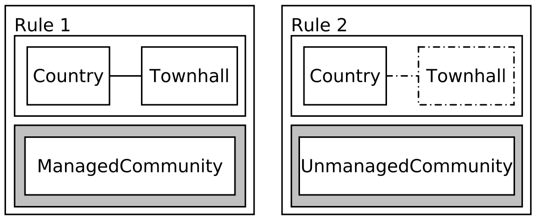


Figure 21: Two rules which may execute on the same input graph.

It is a tempting thought that these two rules would not be able to execute on the same input model, and therefore could not be present in the same path condition. However, it is trivial to construct an input model with two *Country* elements, one connected to a *TownHall* and one not connected. In this case, both rules would execute.

Nonetheless, future work will explore this area more fully to determine if there are conditions where a rule can be said to definitively not execute based on the presence of negative elements.

7.3.3 Creating the Partial Mass Production Sets

We construct the set of partial mass productions $p^*_{partial}$ through the application of mass production construction on each rule combination in the powerset.

Recall that this mass production is simply composed of the matchers and rewriters of the rules in this set. The matcher for the partial combination case will be empty, as we intend for the elements of the rule to be added to the path condition, without binding to any part of the existing path condition.

Also note that unlike the matchers and rewriters for rule application explained in Oakes *et al.* [45], negative elements are retained in the rewriter of each rule as the negative elements must also be stored in the path condition.

7.3.4 Creation of Path Condition Set

The steps above take a path condition *pc* and a set of rules, and produce a total mass production p^*_{total}, and a set of partial mass productions $p^*_{partial}$.

To create the new set of path conditions PC' to be presented to the next layer, we must apply the above productions to the path condition.

Algorithm 1 describes how these productions are applied to a copy of each path condition in the set PC.

```
newPCSet = {}
foreach pc in PC:
 totalRuleSet = findTotalMatches(pc, rules)
 totalMassProd = createMassProd(totalRuleSet)

 pc.apply(totalMassProd)

 partialRuleSet = findPartialMatches(pc, rules)
 partialPowerSet = createPowerSet(partialRuleSet)

 foreach partialCombo in partialPowerSet:
  if isValid(partialCombo):
      p = createMassProd(partialCombo)
      pcCopy = copy(pc)
      pcCopy.apply(p)
      newPCSet.add(pcCopy)

return newPCSet
```

Listing 1: Algorithm for building further path conditions

7.4 Considering Further Rules

Section 7.3 describes how multiple path conditions are created for the combination of a set of path conditions with a set of rules for a layer. This working set of path conditions obtained is then itself combined with the rules in the next layer, to produce yet another set of path conditions. This process will then continue in this layer-by-layer fashion through the transformation.

After all layers have been processed, the path condition set of the last layer contains all the possible path conditions of the transformation. Through our abstraction relation defined in Section 6, this set of path conditions will represent every feasible transformation execution. Section 9 will discuss how our algorithm proves contracts on these path conditions, and thus on all executions of the transformation.

Attribute Setting The setting of attributes for rule elements is also symbolically executed in path condition construction. In our implementation of path condition construction, we store in the path condition the equations stating the values for the attributes as they are assumed by the rules. In subsequent rules, we then check for value compatibility of the match elements being matched. If the conditions on the attributes on the path condition element and the rule element are conflicting, no path condition is generated.

Note that a fairly trivial solver is currently used, as the only available attribute data type in DSLTrans is *String*. However, our approach is not overly restricted by this approach, as we note that our industrial case studies only manipulate *Strings* [53, 52].

More generally, DSLTrans has been used to write many useful transformations with models that have only *Strings* as attributes. This is because DSLTrans specializes in language translations, as described in [41]. Model transformations of this kind typically do not require complex computations over attributes and the bulk of the work is achieved by node matching and rewriting. *String* attribute data is mostly copied over to the generated model or concatenated with other *String* data.

8 Representation and Coverage of the Technique

This section presents our arguments that our path condition building algorithm builds path conditions that both *represent* and *cover* the infinite set of transformation executions.

Representation means that each path condition represents at least one valid transformation execution.

Coverage means that every transformation execution is abstracted by at least one path condition.

These properties are essential for our contract-proving technique as discussed in Section 9. In particular, these properties ensure that our building of path conditions is proper, that the construction reflects the semantics of DSLTrans as discussed in Oakes *et al.* [45], and that the contract proving approach in Section 9 is valid.

8.1 Representation

Proposition 1. *Every path condition abstracts at least one valid transformation execution.*

Proof Sketch. Let $tr \in \textsc{Transforms}$ be a DSLTrans transformation. We wish to demonstrate that for all path conditions, there exists a transformation execution that the path condition abstracts through the abstraction relation. That is: $\forall pc \in \textsc{Pathconds} : (\exists ex \in \textsc{Execs} \mid pc \mapsto ex)$

This representation property will be proved by considering the addition of rules into a DSLTrans transformation. This explanation will follow the pattern of construction in Section 7, and the DSLTrans semantics presented in Oakes *et al.* [45].

8.1.1 Base case:

The base case is when tr is the empty transformation. In this case, as Section 7.1 states, we have only the empty path condition ϵ_{pc} existing in the path condition set.

Recall that Section 6.6 demonstrates that the empty path condition abstracts the empty transformation execution, as well as any execution for which no rule executes. This is due to no input elements or traceability links in the path condition missing in the transformation execution, and no output elements or traceability links missing in the path condition.

8.1.2 Inductive case:

Assuming every path condition generated for a transformation tr abstracts at least one transformation execution, our inductive step will show that every path condition generated from adding a rule to a layer $l \in \text{Layer}$ to tr will also abstract at least one transformation execution. Note that the new transformation execution abstraction will not necessarily be the same as the old.

As discussed in Section 7.2, path conditions are built by examining the rules in a layer l. In the argument that follows, we examine the cases when rules are incrementally added to the layer l.

Note that in the trivial case, the layer l is empty and contains no rules, and the property holds as no new path condition is added to the set of path conditions generated for the transformation tr.

Otherwise, we show that the representation property will also hold when a new rule rl is added to the layer l. We will thus need to consider the three cases of rule combination, depending on the rule's dependencies:

1. Rule rl cannot execute
2. Rule rl has no dependencies or may execute
3. Rule rl will execute

The property trivially holds for Case 1 given that no new path conditions are added to the path condition set in this case.

When Cases 2 or 3 occur, new path conditions are added to the path condition set. Both cases are based on combining a path condition with a rule, as laid out in Section 7.2.

To demonstrate that the property still holds, we show that, whenever the representation property holds for a path condition pc before the rule is added to the transformation, it will hold after.

We start by picking for pc an execution ex such that pc abstracts ex, which exists by our induction hypothesis. Another transformation execution ex' is then built by adding input elements such that the rule rl executes. Let us now demonstrate ex' is abstracted by the path condition pc' where pc' is formed by combining the rule rl with pc, and executing it in ex.

Below we examine how the addition of the rule affects the conditions of the abstraction relation explained in Section 6.4.1:

1. a morphism must exist between the match parts of all the rule copies in the path condition and the input of the execution

2. a surjective morphism must exist between the output of the execution and the apply part of the path condition
3. an injective morphism must exist between the symbolic traceability links of the path condition and the transformation execution
4. a surjective morphism must exist between the traceability links in the the execution and the path condition
5. the morphisms for the above conditions must be consistent with one another

Our arguments are based on the fact that if the morphisms hold in the above conditions, then they will also hold after the addition of the rule elements. Note that our argumentation are applicable for both total combination and partial combination of the rule with the path condition. Both additions add the elements from the rule, without removing any.

Let us start by arguing for why Condition 1 holds for pc' and ex'. As elements present in the matcher for rl are present in pc' through the rule combination, pc then abstracts over some execution ex' where the rule rl has executed on the input model. Note that an injective morphism can then be found between the added matcher elements in pc' and the added input elements in ex'.

As well, traceability links that have been created in the rewriter of the rule will be added to ex' during rule execution. These traceability links will be reflected in the path condition, with the rule being combined with the path condition through the double-pushout approach (Section 7.3.4). Thus, Condition 2 is satisfied.

The argument for Condition 3 follows the argument for Condition 1. As the execution of the rule rl in ex' has produced output elements, these elements must be represented in the path condition pc' through a surjection relation. Again, we know that by our hypothesis that a surjective typed graph isomorphism exists between the output of ex' and the apply part of pc'. 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 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'.

Condition 4 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.

Finally, Condition 5 refers to the need for the binding of all morphisms to be consistent. If the morphisms hold before the rule is added, it is easy to see that the same rule added to both pc' and ex' without structural changes will not change the ability for the morphisms to bind. □

8.2 Coverage

Proposition 2. *(Coverage) Every transformation execution is abstracted by at least one path condition.*

Proof Sketch. Let $tr \in \textsc{Transf}$ be a DSLTans transformation. We wish to demonstrate that, for all transformation executions, there exists a path condition pc such that ex is abstracted by pc, as formally expressed in Section 6.4.1. That is:

$\forall ex \in \textsc{Execs} : (\exists pc \in \textsc{PathConds} \mid pc \mapsto ex)$

This *coverage* can be shown as a corollary of Proposition 1 about the *representation* property of path conditions on transformation executions.

First let us discuss the set of transformation executions where no rules execute. These transformation executions can consist of either an empty input model, or an input model such that no rule matches. As discussed in Section 6.6, these transformation executions are represented by the empty path condition ϵ_{pc}.

In the other case, we consider transformation executions where a set of transformation rules have executed. To assert that these transformation executions are covered by at least one path condition, we will examine the possibility of a counter-example. Therefore, this is a proof by contradiction.

Consider a transformation execution *ex* where there exists no path condition *pc* which abstracts it. There are a limited number of possibilities where this situation may arise:

1. Elements are present in all the path condition input models which are not present in the execution's input
2. Elements are present in the output model but not present in all the path condition output models
3. A rule was not symbolically executed during path condition construction
4. A rule was not symbolically executed the appropriate number of times
5. The abstraction relation is such that the execution cannot be abstracted by any path condition

For the first case, we argue that the input models for all the path condition must be a combination of the input models for the rules in the transformation, including the empty path condition. Therefore, there must be at least one path condition which contains fewer input elements than are present in the execution.

The second case regards the output elements, which in the execution of DSLTrans must have been created by the execution of rules. Therefore, as long as there exists a path condition which captures the executions of the rules, then there cannot be extra output elements that are not covered by a path condition.

The third case goes to the heart of our technique. Note that during path condition construction (Section 7) that our technique is not conservative about building path conditions. This is correct, as the set of path conditions must cover every possible transformation execution.

In particular, we only consider a rule to not symbolically execute on a path condition if the rule enforces dependencies which are not found in the path condition. This is correct, as these dependencies must be satisfied for the rule to execute, but for that particular symbolic execution the elements are not present. In all other cases, the rule is considered to potentially (or definitely) execute, and multiple path conditions are generated with that rule symbolically applied.

The fourth case discusses rule multiplicity. Note that as discussed in Section 7.3.2, rules may need to be symbolically applied multiple times for following rules to be executed. We note in that section that it is possible to determine how many times a rule needs to be duplicated in the path condition construction process. This allows our technique to correctly symbolically apply rules that depend on the multiple execution of other rules. Therefore, for the fourth case, we assume that all rules have been duplicated an appropriate number of times to create a suitable path condition.

Finally, the fifth case raises the question that the abstraction relation does not sufficiently allow for a path condition to abstract the execution. We argue that our abstraction relation is intertwined with the building of each path condition, such that symbolically executing a rule in a path condition directly mirrors the execution of a rule in a transformation execution, following the semantics of DSLTrans defined in Oakes *et al.* [45]. This replication of DSLTrans' semantics mirrors Proposition 1, where a path condition will always abstract at least one transformation execution.

□

9 Verifying Contracts

The algorithm presented in Section 7 produces all possible path conditions for a particular DSLTrans transformation, representing all possible transformation executions. This section will detail another contribution: a method to prove structural contracts on the transformation by examining the set of path conditions to check whether the contracts hold on them.

The abstraction relation presented in Section 6 can then be used to extend the contract proof result to the infinite set of transformation executions. A positive result for contract proof means that the contract will hold on all abstracted transformation executions, while a negative result means that the contract will fail on at least one abstracted transformation execution.

9.1 Structure of a Contract

This section details the structure of a contract in Definition 16. Note that a contract intentionally uses a similar structure to DSLTrans rules. However, in contracts we also allow the possibility of using indirect links in both the input and output graphs.

As well, we must forbid the use of negative elements in contracts. As explained in Section 10.1.1, this is due to the abstraction relation representing a lower bound in abstracted transformation executions.

Definition 16. *Transformation Contract*

Let $tr \in \textsc{Transforms}$ *be a DSLTrans transformation. A contract* c *for* tr *is a four-tuple* $\langle Input, Output, back_{pre}, back_{post} \rangle$*, where:*

- $Input, Output \in \mathrm{TG}$
- *Input and Output may be empty and are disjoint*
- $back = \{E_{back_{pre}}, E_{back_{post}}, (s_{back}, t_{back})\}$
  - $E_{back_{pre}}$ *and* $E_{back_{post}}$ *contain the backward links for the contract*
  - $E_{back_{pre}}$ *and* $E_{back_{post}}$ *are disjoint from* E_{Input} *and* E_{Output}
  - (s_{back}, t_{back}) *is a pair of functions* $s_{back} : E_{back_{pre}} \cup E_{back_{post}} \to V_{Output}$ *and* $t_{back} : E_{back_{pre}} \cup E_{back_{post}} \to V_{Input}$ *that respectively provide the source and target vertices for each backward link*

- $\forall v \in V : isNegativeVertex(v) = false$
- $\forall e \in E : isNegativeEdge(e) = false$

Note that in a contract we have two types of backward links to check for the creation of elements in the path condition. The first type of backward link ($E_{back_{pre}}$) are those backward links that must be present in the pre-condition for the contract to hold. The second type of backward links ($E_{back_{post}}$) will be present in the post-condition of the contract.

The two contracts in Figure 6 contain the first type of backward links.

We define a (trivial) utility function getTransformation: CONTRACT → TRANSFORMS. *This function returns the transformation that the contract is defined for. The purpose of this function is to restrict contracts to only be applicable for a particular transformation. In this case, getTransformation(c) must return tr.*

9.2 Contract Satisfaction

The section will detail the conditions whereby a path condition and transformation execution can be said to satisfy a contract.

First, we create the matchers for the pre-condition and post-condition of the contract. Then, we examine how a contract holds on a transformation execution in Section 9.2.2, and on a path condition in Section 9.2.3.

9.2.1 Contract Matchers

To prove the contract on a transformation execution or a path condition, we need to create matchers for the contract pre-condition and post-condition. These structure are created from the contract structure defined in Definition 16.

Recall that a contract c is a four-tuple $\langle Input,\ Output,\ back_{pre},\ back_{post}\rangle$.

Definition 17. *Contract Pre-condition Matcher*

We define c's pre-condition matcher, noted $\lceil c \rceil_{pre}$, to be a typed graph six-tuple $\langle V, E, (s,t), \tau, VT, ET\rangle$, where:

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

The pre-condition for the contract is created in Definition 17. It consists of the elements in the *Input* graph of the contract, as well as the backward links marked *pre* and attached *Output* elements.

Definition 18. *Contract Post-condition Matcher*

We define c's post-condition matcher, noted $\lceil c \rceil_{post}$*, to be a typed graph six-tuple* $\langle V, E, (s,t), \tau, VT, ET \rangle$*, where:*

- $V = V_{Input} \cup V_{Output}$
- $E = E_{Input} \cup E_{Output} \cup E_{back_{post}}$
- $s = s_{Input} \cup s_{Output} \cup s_{back}$, $t = t_{Input} \cup t_{Output} \cup t_{back}$
- $\tau = \tau_{Input} \cup \tau_{Output}$
- $VT = VT_{Input} \cup VT_{Output}$
- $ET = ET_{Input} \cup ET_{Output}$

The post-condition for the contract is created in Definition 18. It consists of all *Input* and *Output* elements from the contract, along with all backward links marked as *post*.

9.2.2 Transformation Execution Satisfaction

Given the matchers for the contract defined in Definitions 17 and 18, we detail here how a transformation execution satisfies a contract.

Definition 19. *Contract Satisfaction by a Transformation Execution*

Let c be a contract in $\textsc{Contracts}(tr)$ *and ex be a transformation execution in* $Exec(tr)$*. Execution ex satisfies contract c, written* $ex \models c$*, iff:*

$$\exists f, g \mid (\lceil c \rceil_{pre} \underset{f}{\blacktriangleright} ex \wedge \lceil c \rceil_{post} \underset{g}{\blacktriangleright} ex) \wedge$$

f and g are consistent over the same elements

Definition 19 states that a contract holds on a transformation execution if the contract's pre--condition and post-condition matcher hold on the transformation execution through two consistent morphisms.

9.2.3 Path Condition Satisfaction

As there exists an infinite amount of possible transformation executions, proving the contracts directly on the set of transformation executions is not possible. Our technique thus relies on the finite set of path conditions we create to prove contracts on all transformation execution.

Definition 20. *Contract Satisfaction for a Path Condition*

Let c be a contract in $\textsc{Contracts}(tr)$ *and pc be a path condition in PathConds(tr). Path condition pc satisfies contract c, written* $pc \models c$*, iff:*

$$\exists f, g \mid (\lceil c \rceil_{pre} \underset{f}{\rhd} pc \wedge \lceil c \rceil_{post} \underset{g}{\rhd} pc) \wedge$$

f and g are consistent over the same elements

The principle behind the satisfaction relation in Definition 20 is similar to the satisfaction relation between a contract and an execution of a transformation in Definition 19. Whenever the contract's pre-condition matcher is found in the path condition, then satisfaction is based on whether the post-condition matcher of the contract is also found.

Note that despite their similarity the morphisms are different in Definition 19 and Definition 20.

When checking for satisfaction of a contract on a transformation execution, we look for a homomorphism as defined in Definition 4 to match the structures found in the contract.

However, when checking for satisfaction of a contract in a path condition, we instead use a typed graph split morphism, as defined in Definition 5. This allows for elements in the contract to "split" over multiple elements in the path condition.

For example, consider the *Neg-DaughterMother* contract in Figure 6b and the path condition in Figure 7. The pre-condition of the contract cannot be (isomorphically) matched in the path condition, meanwhile the path condition represents the transformation execution where only one *Family* element exists through the abstraction relation. This is thus the motivation for the creation and employment of the split morphism.

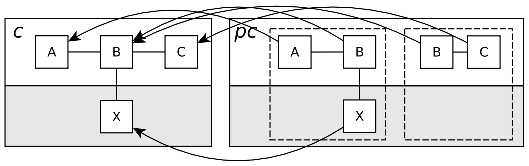


Figure 22: Matching elements between a contract and a path condition using the split morphism.

9.3 Verifying the Transformation

As previously stated, contracts are proved on each path condition created by the path condition generation algorithm. These path conditions are built to symbolically represent all possible executions of that DSLTrans transformation. This proving is done by matching the contract's pre-condition matcher, and potentially the contract's post-condition matcher.

If the pre-condition matcher cannot be found in the path condition, then the pre-condition does not hold for this path condition and the contract will not be checked for this path condition.

If the pre-condition matcher matches, then the post-condition matcher is also checked. If this matching cannot be performed, then the contract does not hold on this path condition, which then serves as a counter-example for the contract. The path condition will contain information on the rules applied, allowing the user to examine the counter-example and reason about the transformation.

Additional Verification Note that in our current implementation of the contract prover, we also perform static checks on the transformation and the contract, such that the required contract elements exist in the transformation, and that rules are symbolically executed an appropriate number of times for the contract to be proved. Our prover can also report if the contract's pre-condition matcher does not match over any path conditions, indicating an invalid or *non-provable* [11] contract.

These checks allow us to determine if there are potential errors with either the contract or the transformation, and have already led to a number of errors being resolved in our industrial case studies.

As well, we have developed initial support for providing detailed information to the user at the end of contract verification. This analysis is designed to assist the user to understanding in which cases the contract holds, and in which cases it does not hold.

During contract proving, we collect a set of path conditions where the contract holds and a set where the contract does not hold. Then, an analysis examines these sets and provides the following information:

- $rules_{good}$ - The rules common to all successful path conditions
- $rules_{bad}$ - The rules common to all failed path conditions, and not common to all successful path conditions
- The precise elements and links that the contract requires from the set of rules $rules_{good}$
- The contract is also tested against a failed path condition
    - The elements and links that could not be found on this path condition are reported

This comprehensive information allows the user to precisely determine how the contract is succeeding or failing, and have assurance that the result is as intended.

10 Contract Proof Validity

Section 8 discusses the *representation* and *coverage* properties of the path condition building technique, while Section 9 examines how contracts are proved on those path conditions. This section takes the next step of proving that this contract proof algorithm will produce *valid* results, as specified in Proposition 3.

Proposition 3. *(Validity) The result of proving a contract on a path condition is the same as proving the contract on all transformation executions abstracted by that path condition.*

Let $tr \in \textsc{Transforms}$ *be a transformation and* $c \in \textsc{Contracts}(tr)$ *be a contract for* tr.

Recall from Section 6 that $pc \mapsto ex$ means that the path condition pc abstracts the transformation execution ex.

Then, $\forall pc \in \textsc{PathConds}(tr)$, the following three implications are satisfied.

10.1 Implication 1 - Positive Satisfaction

$$\forall ex : (pc \mapsto ex \wedge pc \models c) \implies ex \models c \tag{6}$$

Equation 6 states the first implication, which is that if the path condition satisfies the contract, then all abstracted transformation executions must also satisfy the contract.

This can be proven by examining how the contract proof morphisms, which determine how path conditions and transformation executions satisfy the contract, commute (Section 9.2.2 and 9.2.3).

The key insight to our argument is that the path condition will represent the elements that are present in those transformation executions that it abstracts. The examples in Section 6 show how the elements in the path condition must be present in the transformation execution for the abstraction relation to hold. Thus the path condition contains a lower bound on the elements present in the transformation execution.

Multiple applications of the same rule may be present in the transformation execution as well. As noted in Section 7.3.2 we must explicitly denote the number of times each rule symbolically applies in our path condition construction algorithm to create the appropriate path conditions. For this argument, we assume that a static checker has determined this information to allow for proper contract checking.

Based on Definition 19 and Section 6.4.1, the injective matching of the input elements asserts that if the elements of the contract are present in the path condition, then they are also present in the abstracted transformation executions. These morphisms can be easily composed, showing that the elements are present in the transformation execution.

As well, the subjective matching enforces that the elements in the transformation execution are found in the path condition. Note that this subjection handles multiple rule execution. As the path condition represents a lower bound on the number of times the rules have symbolically executed, it is correct to say that if the elements are present in the path condition, then at least that number of elements are present in the abstracted transformation executions.

We can thus compose the morphisms from the contract matching over the path condition, and the abstraction relation from the path condition to all abstracted transformation executions. This allows us to extend the result of contract proving on the path condition to the abstracted transformation executions.

10.1.1 Negative Contract Elements

As discussed in Section 6, the abstraction relation allows for more input elements in the transformation execution than the path condition. This means that other than the elements explicitly disallowed by the negative elements in the path condition, any other element may be present in the transformation execution.

This means that we cannot (currently) support negative elements in the contract. If the contract contains negative elements, and it matches over the path condition, it may still fail on an abstracted transformation execution that contains those elements. Thus, this first implication cannot be satisfied.

Future research will determine if there are cases where negative elements can be allowed in contracts. We have some initial work in propositional joining of contracts, which does have a negation operator [51, 46]. However, the semantics of this operator means that we are verifying that the elements are not *always* present in all abstracted transformation executions.

10.2 Implication 2 - Negative Satisfaction

Equation 7 states that if the contract fails to hold on a path condition, then this implies that the path condition abstracts a transformation execution where that contract will also fail to hold. Thus the path condition will serve as a counter-example to the contract.

$$pc \not\models c \implies \big(\exists ex \mid ex \not\models c \land pc \mapsto ex\big) \tag{7}$$

The transformation execution ex which fails the contract can be easily constructed to be a copy of the path condition pc. That is, the input model of the transformation execution will be composed of the elements present in the input model of the path condition, while the rules executed and the output elements will be the same in both.

Contract satisfaction is specified as a morphism for both satisfaction on transformation executions (Definition 19) and for path conditions (Definition 20). Therefore, if a morphism fails to match the contract on the path condition, it is clear that a morphism will fail to match the contract on the transformation execution.

Note that the typed graph split morphism is used in Definition 20. This morphism is more permissive than a normal morphism as pattern nodes may split over target nodes, allowing for our implication to be true in all cases where a standard isomorphism would hold.

10.3 Implication 3 - Negative Satisfaction from an Execution

Finally, Equation 8 states that if the contract fails to hold on a transformation execution, then the contract must fail to hold on the path condition which abstracts it.

This implication is imperative to ensure that contract verification can correctly examine the set of path conditions to find counter-examples that reflect how the contract holds on the infinite set of transformation executions abstracted.

$$(ex \not\models c \land pc \mapsto ex) \implies pc \not\models c \tag{8}$$

The contract will not hold on the transformation execution if the contract elements cannot be found in the transformation execution, including the input elements, output elements, and backward links. We will argue each of these points, showing how the path condition accurately reflects the structure of the transformation execution.

The input model of the transformation execution may contain more elements than the input graph of the path condition (Section 6.4.1). If the contract matching failed due to missing input elements on the transformation execution, then clearly the elements cannot be present in the path condition.

Assuming the correct multiplicity of symbolic rule execution, the path condition will represent the rules executed in the transformation execution as a lower bound. If the backward links of the contract are not found, then they will not exist in the path condition.

If the output elements of the contract cannot be found in the transformation execution, they will not be present in the path condition. This is due to the subjection relation on backward links in the abstraction relation, which ensures that the same rules that have executed in the transformation execution are also symbolically executed in the path condition.

10.4 Partitioning of Transformation Executions

As explained in Section 7.3.2, our abstraction of rule multiplicities implies a partitioning of the transformation executions. This introduces path conditions that abstract over the same subset of transformation executions, which has consequences for contract proving.

In particular, our contract proving technique must be careful about how this non-unique abstraction and sensitivity to rule multiplicity affects the counter-examples produced.

Let PC_{Super} be a path condition that abstracts one or more applications of rule A, and PC_{Sub} be a path condition that abstracts two or more applications of rule A. In this case, the transformation executions that are abstracted by PC_{Super} are a superset of those abstracted by PC_{Sub}. The core issue discussed here is how to reason about the cases where the result of proving a contract c on PC_{Super} and PC_{Sub} differs.

Recall that in the definition of a contract, we have disallowed negative elements, which is motivated in Section 10.1.1. As well, DSLTrans rules cannot delete elements when applied. Therefore, if a path condition satisfies a contract before application of a rule, then it must still satisfy it after.

Thus the case of interest is when $PC_{Super} \not\models c$ and $PC_{Sub} \models c$. This would be due to the addition of elements required by c which are produced by the repeated application of rule A.

In this case, our contract proving approach would report PC_{Super} as a counter-example to the contract and PC_{Sub} as a set of transformation executions where the contract holds. This is correct reasoning, as the applications of rules represented by PC_{Super} does not satisfy the contract, and further application of the rule is required.

This situation raises a key criticism of our approach. As discussed for path condition construction (Section 7.3.2), our approach must determine the number of times to symbolically apply each rule in the transformation such that a contract can be proven. If a rule is not symbolically applied an appropriate number of times, then a path condition that satisfies the contract will not be constructed.

11 Related Work

According to the classification in [8] the technique presented in this paper deals with contracts structured as *model syntax relations* as described in Section 9. Such contracts check whether certain elements in the input model are necessarily transformed into other elements in 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

UML [7]. 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 [44]. 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 [8]) 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.

The presented approach in this paper allows for contracts to be proved for all possible transformation executions, i.e., for all possible input models. However, we also keep the same implication idea: the pre-condition of a property sets constraints on the input models of the transformation, and then, the post-condition defines constraints on the output model.

In [27, 55] the authors describe their method where 'Tracts' can be specified for model transformations. These tracts define a set of constraints on the source and target metamodels, a set of source-target constraints, and a tract test suite, i.e., a collection of source models satisfying the source constraints. The accompanying TractsTool can then automatically transform the source models into the target metamodel, and subsequently verify that the source/target model pairs satisfy the constraints. The advantages of this are that the approach is not computationally-intensive, as tests can be narrowly focused in a modular way.

Besides the Tracts approach, there are several other approaches supporting the testing of model transformations based on different kind of contracts such as model fragments [43], graph patterns [31, 10], Triple Graph Grammars (TGGs) [60], dedicated testing languages [36, 25], or as used in Tracts OCL constraints [19, 20], and even a combination of these mentioned approaches [26].

In 2009 [19] Cariou *et al.* study the use of OCL contracts in the verification of model transformations. The approach 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 [9]. 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 [38]. 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 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.* [31] 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. However the verification technique used by Guerra *et al.* differs fundamentally from ours. Our abstraction over rule multiplicity enables our approach to be exhaustive, 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 [30].

Also in 2012, Büttner *at al.* have published their work on the verification of ATL transformations [17, 18].

The approach in [17] 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. 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.

There are several differences with our approach. First, the authors' proofs may require human assistance, depending on the used SAT solver. Also, 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 counter-examples 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 a subset of ATL that is verifiable.

In [18] Büttner *at al.* aim at proving model syntax relation properties of ATL transformations expressed as pre-condition/ post-condition OCL constraints. The authors provide 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.

Assertional reasoning in graph transformations has been studied by Habel and colleagues, who have introduced nested conditions as properties of graphs in [32]. The authors formally prove these nested conditions have the expressiveness of first-order graph formulas. Poskitt and Plump later propose in [47] 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.

A different possibility for our work would have been to utilise the GROOVE tool, which can specify, play, and analyse graph transformations [50]. 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 [48] 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. In our work, this explosion issue is dealt with by restricting the number of times each rule is represented in a path condition.

Still in the context of GROOVE, several studies [49, 13, 50] 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 to examine the state they represent. We rather symbolically represent all transformation executions using an abstraction relation with the path conditions. We are therefore 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 [14]. 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.

Another approach to model transformation verification involves semantic analysis of the transformation under study as well as the input and output languages. In Barroca *et al.* [**?**], operational semantics are provided for the input and output languages using SOS. When a DSLTrans transformation is provided which maps the input language constructs to the output language constructs, the approach is able to check the semantics preservation of the transformation and provide counterexamples.

12 Conclusion

In this paper we have presented a symbolic execution technique for the DSLTrans language which builds all possible path conditions for a transformation. We have also presented our contract verification technique, which can prove pre-condition/ post-condition contracts on those path conditions and report counter-examples.

The approach of building path conditions to represent transformation executions is made possible through our abstraction relation. This relation ensures that the elements and rules present in a transformation execution are present in a contract. As well, we have discussed how our technique can represent multiple application of a rule, and the consequences of this for path condition construction and contract proving.

This work and the previous research on formalizing DSLTrans (Oakes *et al.* [45]) places our symbolic execution within the context of the double-pushout approach. Thus we recognize that the DSLTrans language is very similar to other model transformation languages which also rely on graph-rewriting techniques.

Our future work will therefore extend the path condition building and contract proving contributions to these other languages. In particular, it may be sufficient to define a short list of restrictions to be placed on a language to fit our contract proving technique[2]. If these restrictions are met, it may be possible to provide contract verification for the language simply by defining the matchers and rewriters for each rule in the transformation.

The authors would like to deeply thank Dániel Varró, Clark Verbrugge, and the late Bernhard Schätz for their detailed and helpful comments.

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

References

[1] ATL Zoo. `http://www.eclipse.org/atl/atlTransformations`.

[2] Atlas Transformation Language. `http://eclipse.org/atl`.

[3] Eclipse Modelling Framework. `https://eclipse.org/modeling/emf/`.

[4] Eclipse Platform. `https://eclipse.org/`.

[5] Epsilon Generation Language. `http://www.eclipse.org/epsilon/`.

[6] mbeddr. `www.mbeddr.com`.

[7] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In *UML 2002 - The Unified Modeling Language*, pages 243–258. Springer, 2002.

[8] Moussa Amrani, Jüergen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Towards a model transformation intent catalog. In *Proceedings of the First Workshop on Analysis of Model Transformations (AMT)*, October 2012.

[2]An initial examination suggests that these restrictions would include *termination*, *confluence*, *non-deletion*, and *layer-based*.

[9] M. Asztalos, L. Lengyel, and T. Levendovszky. Towards automated, formal verification of model transformations. In *ICST*, pages 15–24. IEEE, 2010.

[10] András Balogh et al. Workflow-driven tool integration using model transformations. In *Graph Transformations and Model-Driven Engineering*, pages 224–248.

[11] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: a Turing incomplete transformation language. In *SLE*, pages 296–305. Springer.

[12] Bruno Fontes Barroca. *Analysable Software Language Translations*. PhD thesis.

[13] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink. A modal-logic based graph abstraction. In *ICGT*, volume 5214 of *Lecture Notes in Computer Science*, pages 321–335. Springer.

[14] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *ICSE*, pages 72–81. ACM.

[15] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A reactive model transformation platform. In *International Conference on Theory and Practice of Model Transformations*, pages 101–110. Springer.

[16] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers.

[17] F. Büttner, M. Egea, and J. Cabot. On verifying ATL transformations using off-the-shelf SMT solvers. In *MoDELS*, pages 432–448. Springer.

[18] F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL transformations using transformation models and model finders. In *ICFEM*, pages 198–213. Springer.

[19] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL contracts for the verification of model transformations. 24.

[20] Robert Clarisó, Jordi Cabot, Esther Guerra, and Juan de Lara. Backwards reasoning for model transformations: Method and applications. 116:113–132.

[21] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159.

[22] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches.

[23] J. De Lara and H. Vangheluwe. Automating the transformation-based analysis of visual languages. 22(3-4):297–326.

[24] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs In Theoretical Computer Science)*. Springer-Verlag New York, Inc.

[25] Antonio García-Domínguez, Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Inmaculada Medina-Bulo. EUnit: a unit testing framework for model management tasks. In *Proc. of MoDELS*, pages 395–409.

[26] Pau Giner and Vicente Pelechano. Test-driven development of model transformations. In *Proc. of MoDELS*, pages 748–752.

[27] Martin Gogolla and Antonio Vallecillo. Tractable model transformation testing. In *Proc. of ECMFA*, pages 221–235.

[28] Cláudio Gomes, Bruno Barroca, and Vasco Amaral. Classification of model transformation tools: Pattern matching techniques. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems SE - 38*, volume 8767 of *Lecture Notes in Computer Science*. Springer International Publishing.

[29] Cláudio Gomes, Bruno Barroca, and Vasco. Amaral. DSLTrans User Manual. `http://msdl.cs.mcgill.ca/people/levi/files/DSLTransManual.pdf`.

[30] Carlos A González and Jordi Cabot. ATLTest: a white-box test generation approach for ATL transformations. In *Model Driven Engineering Languages and Systems*, pages 449–464. Springer.

[31] E. Guerra, J. De Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated verification of model transformations based on visual contracts. 20(1):5–46.

[32] Annegret Habel and Karl-heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. 19(2):245–296.

[33] ISO26262 ISO. 26262: Road vehicles-functional safety. 26262.

[34] Jetbrains. Meta Programming System. `https://www.jetbrains.com/MPS`.

[35] J.C. King. Symbolic execution and program testing. 19(7):385–394.

[36] Dimitrios Kolovos, Richard Paige, Louis Rose, and Fiona Polack. Unit testing model management operations. In *Proc. of ICSTW*, pages 97–104.

[37] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In *International Conference on Model Driven Engineering Languages and Systems*, pages 240–255. Springer.

[38] T. Levendovszky, L. Lengyel, G. Mezei, and H. Charaf. A systematic approach to metamodeling environments and model transformation systems in VMTS. 127(1):65–75.

[39] L. Lúcio, B. Barroca, and V. Amaral. A technique for automatic validation of model transformations. In *MoDELS*, pages 136–150. Springer.

[40] Levi Lúcio, Bentley Oakes, and Hans Vangheluwe. A technique for symbolically verifying properties of graph-based model transformations. Technical report, Technical report, Technical Report SOCS-TR-2014.1, McGill U.

[41] Levi Lúcio, Bentley James Oakes, Cláudio Gomes, Gehan MK Selim, Juergen Dingel, James R Cordy, and Hans Vangheluwe. SyVOLT: Full model transformation verification using contracts. In *P&D@ MoDELS*, pages 24–27.

[42] Lúcio, Levi and Amrani, Moussa and Dingel, Jürgen and Lambers, Leen and Salay, Rick and Selim, Gehan and Syriani, Eugene and Wimmer, Manuel. Model transformation intents and their properties. pages 1–38.

[43] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Model transformation testing: Oracle issue. In *Proc. of ICSTW*, pages 105–112.

[44] A. Narayanan and G. Karsai. Verifying model transformations by structural correspondence. 10.

[45] Bentley James Oakes, Levi Lúcio, Cláudio Gomes, and Hans Vangheluwe. Complete semantics for the DSLTrans transformation language. Technical Report CS-TR-2017.2, McGill University.

[46] Bentley James Oakes, Javier Troya, Levi Lúcio, and Manuel Wimmer. Full contract verification for ATL using symbolic execution. pages 1–35.

[47] Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. 118(1-2):135–175.

[48] A. Rensink, A. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *ICGT*, pages 226–241. Springer.

[49] Arend Rensink and Dino Distefano. Abstract graph transformation. 157(1):39–59.

[50] Arend Rensink and Eduardo Zambon. Pattern-based graph abstraction. In *International Conference on Graph Transformation*, pages 66–80. Springer.

[51] Gehan MK Selim. *Formal Verification of Graph-Based Model Transformations*. PhD thesis.

[52] Gehan MK Selim, James R Cordy, Juergen Dingel, Levi Lúcio, and Bentley J Oakes. Finding and fixing bugs in model transformations with formal verification: An experience report. In *Proc. of AMT*, pages 26–35.

[53] Gehan MK Selim, Levi Lúcio, James R Cordy, Juergen Dingel, and Bentley James Oakes. Specification and verification of graph-based model transformation properties. In *International Conference on Graph Transformation*, pages 113–129. Springer.

[54] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. 20(5):42–45.

[55] Antonio Vallecillo, Martin Gogolla, Loli Burgueno, Manuel Wimmer, and Lars Hamann. Formal specification and testing of model transformations. In *Formal Methods for Model-Driven Engineering*, pages 399–437.

[56] Herman Van der Auweraer, Jan Anthonis, Stijn De Bruyne, and Jan Leuridan. Virtual engineering at work: The challenges for designing mechatronic products. 29(3):389–408.

[57] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, and Hans Vangheluwe. Domain-specific modelling for human–computer interaction. In *The Handbook of Formal Methods in Human-Computer Interaction*, pages 435–463. Springer International Publishing.

[58] Hans Vangheluwe, Juan De Lara, and Pieter J. Mosterman. An introduction to multi-paradigm modelling and simulation. In *Proceedings of AIS2002 (AI, Simulation & Planning)*, pages 9–20. SCS.

[59] D. Varró, S. Varró-Gyapai, H. Ehrig, U. Prange, and G. Taentzer. Termination analysis of model transformations by Petri nets. In ICGT, volume 4178, pages 260–274. Springer.

[60] Martin Wieber, Anthony Anjorin, and Andy Schürr. On the usage of TGGs for automated model transformation testing. In *Proc. of ICMT*, pages 1–16.