A SYMBOLIC EXECUTION-BASED APPROACH TO MODEL TRANSFORMATION VERIFICATION USING STRUCTURAL CONTRACTS

by

Bentley James Oakes

School of Computer Science
McGill University, Montreal, Quebec

August 2018

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

As the complexity of software systems increases, the engineering effort for developing those systems must deal with that complexity. One paradigm for software development is *model-driven engineering*, where the models of the system become the first-order artefacts. These models may be used for simulation or analysis of the system, or be transformed into executable code or documentation. The intention is to represent each facet in the system in the most appropriate formalism at the most appropriate level of abstraction.

Model transformations provide a structured and understandable way of manipulating these models, and are often rooted in a mathematical approach which enables precise specification and analysis. However, it may be difficult for a user to reason about what elements will be matched and written by a particular transformation.

Our research is focused on the verification of model transformations for a particular model transformation language. In particular, we are interested in the proving of pre-condition/ post-condition contracts, which relate the elements in the input models to the transformation with the elements present in the corresponding output elements.

DSLTrans

The DSLTrans transformation language was selected for our research because of the properties guaranteed by construction: *termination* and *confluence*. In this thesis, we provide a description of the semantics of DSLTrans in the double-pushout approach, which brings DSLTrans to the state-of-the-art in model transformation. As well, we provide semantics for other DSLTrans constructs not considered in earlier works. These constructs include negative elements, indirect links, and *Exists* elements.

Contract Proving

The core idea of our contract proving approach is to build *path conditions* for the DSLTrans transformation through a symbolic execution procedure. That is, our technique determines all valid combinations of rules in the transformation. Each of these path conditions will explicitly represent the input and output elements present when that combination of transformation rules execute through an *abstraction relation*.

We then prove pre-condition/ post-condition contracts on these path conditions. If a path condition does not satisfy the contract, then the path condition serves as a *counter-example*. Examination of this counter-example allows for reasoning about rules and rule interaction in the transformation.

As a contribution of this thesis, we consider all constructs in the DSLTrans language and thus we can verify all DSLTrans transformations, including those that contain negative elements. As well, we also define the unique *split morphism* in this thesis, which allows for the vertices in the pattern graph to 'split' over the vertices in the target graph. The use of this morphism allows us to create fewer path conditions than in previous research, enabling our technique to scale to larger transformations.

SyVOLT Tool

Another contribution of this thesis is to present the SyVOLT tool, which is our implementation of the path condition generation and contract proof techniques. In this thesis, we discuss the work-flow and implementation details of the prover, with a focus on discussing the multi-paradigm and multi-formalism nature of the tool. We also present a number of techniques to improve the scalability and analysis results of the tool.

Résumé

À mesure que la complexité des systèmes de logiciels augmente, l'effort d'ingénierie pour développer ces systèmes doit faire face à cette complexité. Un paradigme pour le développement des logiciels est l'ingénierie axée sur le modèle, où les modèles du système deviennent les artéfacts de premier ordre. Ces modèles peuvent être utilisés pour la simulation ou l'analyse de système, ou être transformés en code exécutable ou en documentation. L'intention est de représenter chaque facette dans le système dans le formalisme le plus approprié et au niveau d'abstraction le plus approprié.

Les transformations de modèles fournissent une manière structurée et compréhensible de manipuler ces modèles, et sont souvent ancrées dans une approche mathématique qui permet une spécification et une analyse précises. Cependant, il peut être difficile pour un l'utilisateur de raisonner sur quels éléments seront appariés et créer par une transformation particulière.

Notre recherche se concentre sur la vérification des transformations de modèles pour un langage de transformation de modèle spécifique. En particulier, nous sommes intéressés par prouvant des contrats de pré-condition / post-condition, qui relient les éléments intrants des modèles à la transformation avec les éléments présents dans les éléments d'extrants correspondants.

DSLTrans

Ce langage de transformation DSLTrans a été sélectionné pour la vérification en raison des propriétés garanties par construction : terminaison et confluence. Dans cette thèse, nous fournissons une description de la sémantique de DSLTrans dans l'approche « double-pushout », qui apporte DSLTrans au point dans les transformations des modèles. De plus, nous fournissons les sémantiques pour d'autres constructions DSLTrans qui ne sont pas considérées dans des travaux antérieurs. Ces constructions comprennent des éléments négatifs, liens indirects, et éléments «Existe».

Preuve de Contrat

L'idée de base de notre approche de vérification des contrats est de construire des conditions de trajet pour la transformation DSLTrans à travers une procédure d'exécution symbolique. Autrement dit, notre technique détermine toutes les combinaisons valides de règles dans la transformation. Chacune de ces conditions de trajet représentera explicitement les éléments intrant et extrants présents lorsque cette combinaison des règles de transformation s'exécute à travers une relation d'abstraction.

Nous vérifions ensuite les contrats de précondition / post-condition sur ces conditions de trajet. Si une condition de trajet ne satisfait pas le contrat, la condition de trajet sert de contrexemple. L'examen de ce contrexemple permet de raisonner sur les règles et l'interaction des règles dans la transformation.

En tant que contribution de cette thèse, nous considérons toutes les constructions dans le langage DSLTrans et ainsi nous pouvons vérifier toutes les transformations DSLTrans, y compris celles qui contiennent des éléments négatifs. De plus, dans cette thèse, nous définissons également l'unique morphisme de division, ce qui permet aux sommets dans le graphique de modèle de se «diviser» sur les sommets dans le graphique cible. L'utilisation de ce morphisme nous permet de créer moins de conditions de trajets que dans les recherches précédentes, permettant à notre technique d'évoluer vers de plus grandes transformations.

SyVOLT Outil

Une autre contribution de cette thèse est de présenter l'outil SyVOLT, qui est notre implémentation des techniques de génération de condition de trajet et de preuve de contrat. Dans cette thèse, nous discutons de la procédure et des détails de mise en œuvre de l'outil, en mettant l'accent sur la nature multiparadigme et multiformalisme de l'outil. Nous présentons également un certain nombre de techniques pour améliorer l'évolutivité et l'analyse des résultats de l'outil.

Acknowledgements

I would like to thank all those who have guided me in my life, especially my friends and colleagues. Every little bit of help has been instrumental in all my adventures.

For my supervisors, Clark and Hans, thank you for giving me this opportunity. For Levi Lúcio, thank you for the wonderful time in Munich and for being a friend as well as a colleague. I also deeply appreciate the insights by Dániel Varró, Juan de Lara, and Esther Guerra.

I acknowledge the generous support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

I thank my family for being there for me and listening to me. For my parents and brother, I will always be there for you.

And for my wonderful wife, thank you for being the amazing person that you are. I cannot put into words how much you mean to me, Alicia. And all of love to our little V/B.

Publications

This research has resulted in the following publications. Please note that Section 1.3 discusses the author's specific contributions to these publications.

1. **B. Oakes**, C. Verbrugge, L. Lúcio,and H. Vangheluwe. Debugging of Model Transformations and Contracts in SyVOLT. Accepted in the Debugging in Model-Driven Engineering (MDEbug 2018) workshop.
2. **B. Oakes**, L. Lúcio, C. Gomes, and H. Vangheluwe. Expressive Symbolic-Execution Contract Proving for the DSLTrans Transformation Language. Technical Report SOCS-TR-2017.1, McGill University, 2017.
3. **B. Oakes**, J. Troya, L. Lúcio, and M. Wimmer. Full Contract Verification for ATL using Symbolic Execution. *Software and Systems Modeling*, pages 1–35, 2016.
4. **B. Oakes**, J. Troya, L. Lúcio, and M. Wimmer. Fully Verifying Transformation Contracts for Declarative ATL. In *International Conference on Model Driven Engineering Languages and Systems*, pages 256–265, 2015.
5. L. Lúcio, **B. Oakes**, C. Gomes, G. Selim, J. Dingel, J. Cordy, and H. Vangheluwe. *SyVOLT: Full Model Transformation Verification using Contracts.* In *International Conference on Model Driven Engineering Languages and Systems*, pages 24–27, 2015.

6. G. Selim, J. Cordy, J. Dingel, L. Lúcio, and **B. Oakes**. *Finding and Fixing Bugs in Model Transformations with Formal Verification: An Experience Report.* In *Proceedings of Analysis of Model Transformations workshop at Model Driven Engineering Languages and Systems*, pages 26–35, 2015.
7. L. Lúcio, **B. Oakes**, and H. Vangheluwe. *A Technique for Symbolically Verifying Properties of Graph-based Model Transformations.* Technical Report SOCS-TR-2014.1, McGill University, 2014.
8. G. Selim, J. Cordy, J. Dingel, L. Lúcio, and **B. Oakes**. *Specification and Verification of Graph-Based Model Transformation Properties.* In *Proceedings of International Conference on Graph Transformation*, pages 113–129, 2014.

Contents

1 Introduction **2**

1.1 Problem Overview 4

1.1.1 Research Questions 5

1.2 Solution Overview and Contributions 5

1.2.1 DSLTrans Semantics in the Double-Pushout Approach 6

1.2.2 Structural Contract Proving 7

1.2.3 Prover Implementation 12

1.3 Author Contributions 14

1.4 Thesis Organization 17

2 Background **19**

2.1 Model Transformations 19

2.1.1 Transformation Rules 21

2.2 DSLTrans Introduction 22

2.2.1 Families-to-Persons Transformation 23

2.2.2 Rules 26

2.2.3 Layers 27

2.2.4 Backward Links 28

2.2.5 DSLTrans Constructs 30

2.2.6 Termination and Confluence Properties 32

2.3 Fundamental Definitions 33
2.3.1 Graphs 33
2.3.2 Graph Morphisms 35

3 DSLTrans Formalization 42
3.1 DSLTrans Structures 43
3.1.1 Transformation Rule 43
3.1.2 Layer and Transformation 45
3.1.3 Input-Output Model 46
3.1.4 Matcher and Rewriter 47
3.2 DSLTrans Semantics 51
3.2.1 Application of a DSLTrans Rule 51
3.2.2 Negative Application Conditions 56
3.2.3 Attributes 59
3.2.4 Indirect Links 61
3.2.5 Repeated Application of a DSLTrans Rule 63
3.2.6 Any versus Exists Elements 65
3.2.7 DSLTrans Layer Execution 70
3.2.8 DSLTrans Transformation Execution 71
3.3 Complexity of DSLTrans 72
3.4 Confluence and Termination Properties 74
3.4.1 Confluence 74
3.4.2 Termination 77
3.5 Conclusion 78

4 Transformation Verification using Contracts 80
4.1 Abstraction Relation 82
4.1.1 Symbolic Execution 82
4.1.2 Rule Combinations 83
4.1.3 Path Conditions 85

4.1.4 Definition of the Abstraction Relation 87
4.1.5 Examples 92
4.1.6 Other DSLTrans Constructs 103
4.1.7 Partitioning of Input-Output Models 103
4.2 Building Path Conditions 104
4.2.1 Path Condition Generation Overview 104
4.2.2 Rule Classification Sets 106
4.2.3 Rule Combination 111
4.2.4 Considering Further Layers 118
4.3 Representation and Coverage of the Technique 118
4.3.1 Representation 119
4.3.2 Coverage 123
4.4 Contract Proof 126
4.4.1 Structure of a Contract 126
4.4.2 Contract Satisfaction 130
4.4.3 Transformation Verification 134
4.5 Validity of Contract Proof 136
4.5.1 Implication 1 - Positive Satisfaction 137
4.5.2 Implication 2 - Negative Satisfaction from Path Condition . . 139
4.5.3 Implication 3 - Negative Satisfaction from an Input-Output Model 140
4.5.4 Partitioning of Transformation Executions 142
4.6 Contract Expressiveness 143
4.6.1 Attributes 143
4.6.2 Propositional Logic and Pivots 144
4.6.3 Types of Contracts 151
4.6.4 Limitations 153
4.7 Conclusion 155

5 SyVOLT Design and Work-flow 157
5.1 FTG+PM . . . 158
5.2 Transformation and Contract Construction . . . 163
5.2.1 Higher-order Transformation from ATL . . . 164
5.2.2 Eclipse Plugin . . . 169
5.2.3 Meta Programming System (MPS) . . . 171
5.3 Proving Process Implementation . . . 176
5.3.1 Himesis Format . . . 176
5.3.2 T-Core . . . 180
5.3.3 PyRamify . . . 183
5.3.4 Path Condition Creation . . . 188
5.3.5 Contract Proof . . . 189
5.4 Conclusion . . . 190

6 SyVOLT Algorithms 192
6.1 Experimental Conditions . . . 193
6.2 Split Morphism Matching Algorithm . . . 194
6.2.1 Step 1 - Candidate Generation . . . 198
6.2.2 Typing and Attribute Resolution . . . 201
6.2.3 Step 2 - Match Set Creation . . . 202
6.2.4 Complexity Analysis . . . 207
6.2.5 Experimental Results . . . 210
6.3 Algorithmic Optimizations . . . 213
6.3.1 Parallelization . . . 214
6.3.2 Slicing . . . 222
6.3.3 Pruning . . . 229
6.3.4 Conclusion . . . 235
6.4 Checks and Analysis . . . 237
6.4.1 Dependency Checking . . . 238
6.4.2 Rule Application . . . 241

6.4.3 Contract Result Analysis 241
6.5 Conclusion 247

7 Case Studies 248
7.0.1 Case Study Sizes 248
7.0.2 Contract Description 249
7.1 RSS2ATOM 251
7.1.1 Transformation 251
7.1.2 Contracts 253
7.2 GM-to-AUTOSAR Transformation 255
7.2.1 M6 Contract 255
7.3 UML-to-Kiltera Transformation 256
7.3.1 Contracts 257
7.4 Families-to-Person Transformation 259
7.4.1 Contracts 263
7.5 mbeddr 270
7.5.1 Transformation 270
7.5.2 Contract 273

8 Related Work 276
8.1 DSLTrans 276
8.1.1 Graph Transformation Formalism 276
8.1.2 In-place versus Out-place 277
8.1.3 Control Flow 278
8.1.4 Confluence and Termination 278
8.2 Model Transformation Verification 279
8.2.1 State-space Analysis 280
8.2.2 Contract Proving 281
8.3 SyVOLT Algorithms 284
8.3.1 Contract/Transformation Analysis 286

9 Conclusion and Future Work **288**

9.1 Research Questions 289

9.1.1 Formalization of DSLTrans 289

9.1.2 Formalization of the Contract Proving Approach 290

9.1.3 Design of SyVOLT 291

9.1.4 Algorithmic Scalability Techniques 292

9.2 Future Work 292

9.2.1 Transfer Contract Proving Approach 293

9.2.2 Creation Process for Contracts 293

9.2.3 Integration with Modelling Tools 294

Chapter 1
Introduction

Over the past few years, software has become an integral part of daily life, from smart thermometers [168] to the industrial automotive domain [138]. As these systems become more inter-connected and feature-rich, there is a increasing demand to effectively manage the complexity in the system during its construction, maintenance, and analysis stages [55].

Model-driven engineering (MDE) [133] is a method to manage the complexity in the systems under study by using models at an appropriate level of *abstraction*. The intention is to separate the concepts in the problem space ('what problem is to be solved') from the concepts in the solution space ('how the problem is to be solved') [161]. Thus, the user is afforded appropriate complexity without introducing accidental complexity.

Once models of the system have been created at an appropriate of abstraction, the models can be fed as input into tools to *simulate* the model, or a solution (such as code) may be generated directly from the model. This use of MDE has been highly successful and is increasingly employed in the engineering of physical systems [156].

A core tenet of model-driven engineering is that the models are *typed* by a *meta-model* [84]. This meta-model is the language for the model which defines the constraints and types available to write the model in. For example, a class diagram which is itself written in the Unified Modelling Language [32] will define the classes available for instantiation in a model. Thus, the model is said to conform to or be typed by the classes in the meta-model [27]. This relation is discussed in Section 2.1 when we discuss transformations further. In this thesis, we will also refer to a model as being *in a language*, or the meta-model as the *model's language*.

One approach to managing model complexity is through the creation of *domain-specific modelling languages* for the meta-model, such that the model's language encodes concepts from the problem domain [79]. This allows domain experts to reason about their model more efficiently, without having to become experts in the solution space [162]. This has been seen to increase productivity significantly [76, 78]. An example of domain-specific modelling is seen in Van Mierlo *et al.* [159] where the authors present a nuclear power system using an intuitive visual syntax.

Model transformations are a key part of the MDE method, and have even been called "the heart and soul of model-driven software development" [140]. The reason for the adoption of transformations is that they can manipulate models in a structured and repeated way with an excellent compromise between strong theoretical foundations and applicability to real-world problems [34].

For example, both a particular model and its meta-model can be represented as a typed attributed graph, which can be modified by a transformation [84]. Transformations may perform any number of operations on the model and meta-model, such as creating, updating, or deleting model elements [45, 64]. Thus, transformations can operate on the high-level domain-specific concepts in the model language, while still allowing for mathematical treatment based on the foundations of graphs and graph transformations.

There may be different intents for these transformations [93], such as *simulation* to represent a system changing over time [47], *synthesis*, where for example executable code is created from the model, or as a *migration* to change the language of the model while preserving the content [137, 135].

More detail about the interaction between transformations, models, and meta-models is found in the background material for this thesis (Section 2.1).

1.1 Problem Overview

As adoption of MDE increases in both academia and industry, the verification of these transformations becomes a critical part of software development. For example, a code generation transformation which converts a software system model into executable code may be subject to industry standards. The ISO-26262 [73] standard in the automotive domain requires documentation and assertions that the produced artefacts are correct.

Properties of interest for a transformation may be that the transformation always terminates or can never produce an unsafe model where a safety property does not hold. In this thesis, we focus on the proving of *structural contracts*, such that if a pre-condition holds on the input model, then a post-condition will hold on the output model. That is, our structural contracts provide proof that a relationship exists or does not exist between input and output elements. For example, the contract in Figure 1.2a on page 9 relates members of a family in the input model with the male and female elements produced in the output model.

One technique for verifying a transformation is to explore all possible outcomes of the transformation, which is termed *state-space exploration* or *symbolic execution* (related work is discussed in Section 8.2.1).

However, this verification technique may be infeasible due to a number of issues which cause an explosion of possibilities in the state space. The first issue is that a transformation under study may contain many rules and elements, making it difficult to build all possibilities of interactions between the rules. Second, this explosion of possibilities may be made much worse if the order of rule application changes the final result. Finally, if the transformation is not guaranteed to terminate and the resulting state-space is infinite, then it may be impossible to verify the transformation. We address these issues with our symbolic execution technique.

1.1.1 Research Questions

Our thesis statement is thus:

How can structural contracts be efficiently verified on a domain-specific transformation language, given certain expressiveness restrictions?

We further break down this thesis statement into a collection of research questions (RQs), which will be answered throughout this thesis:

— RQ1) *How can the semantics of a model transformation language with reduced expressiveness (DSLTrans) be precisely formalized and brought to the state-of-the-art?*

— RQ2) *How can the infinite execution possibilities of a transformation be represented in an explicit finite set?*

— RQ3)

  — a) *How can structural pre-condition/ post-condition contracts be proven to be satisfied or non-satisfied on these representations?*

  — b) *When a contract is not satisfied, how do the counter-examples produced relate to the transformation?*

— RQ4) *What is the design and work-flow of a contract verification tool?*

— RQ5) *What are techniques for improving the scalability of the verification tool to allow for larger transformations to be verified?*

The next section will present an overview of our solutions to the above research questions.

1.2 Solution Overview and Contributions

Our research is focused on techniques for the creation of a symbolic execution state-space for transformations, and verification of certain properties over that representation of the state-space. We employ three components in our solution to this problem: the DSLTrans language, the contract proving approach, and the SyVOLT tool which implements this contract proving approach.

1.2.1 DSLTrans Semantics in the Double-Pushout Approach

To avoid the mentioned problem of an explosion in state-space, we require that the transformation language being verified have reduced expressiveness, as in work by Varró *et al.* [166]. Our research verifies the model transformation language DSLTrans, which has the properties of *termination* and *confluence*. Termination means that transformations will execute in a finite amount of time, while confluence means that transformations always produce the same output model for the same input model. This ensures that the state-space is finite, and lowers the number of state-spaces that have to be created.

Note that although the language has restricted expressiveness, our research and case studies show that DSLTrans is applicable to real transformation problems which involve *translation* or *migration* transformations [98, 135, 113]. These translation transformations translate a model in one meta-model into another meta-model, which may be done to 'evolve' the model to a new meta-model. An example of this is our industrial case study *GM-to-AUTOSAR* (Section 7.2) where the input model is modelled in a proprietary language, and is translated into an industry standard language for use with a greater variety of tools.

1.2.1.1 Contributions

The DSLTrans transformation language used in this thesis was introduced in past work [21, 19, 94] and is thus not a contribution of this thesis. However, the past descriptions of the semantics of DSLTrans have been incomplete.

The first contribution of this thesis is to formalize the semantics of DSLTrans in Chapter 3 by placing the language within the double-pushout approach. This intuitive approach provides greater clarity to the use and expressiveness of DSLTrans. As well, by aligning DSLTrans closer with the current state-of-the-art, future work will be able to extend the contract verification approach to other model transformation languages.

Second, this thesis presents significant advances over earlier works by formalizing a number of DSLTrans constructs not addressed in earlier works. These constructs include *Any* and *Exists* elements, indirect links, and the negative versions of elements which prevents rule application, which enables more expressiveness for DSLTrans transformations.

The formalization of the semantics of DSLTrans is also required for our approach to contract proving. This is made concrete in Section 4.2, where we represent the application of DSLTrans rules. The complete formalization of all constructs of DSLTrans allows us to consider them in our contract proving approach, and we can now state that our contract prover technique handles all DSLTrans transformations.

1.2.2 Structural Contract Proving

Our contract verification technique (discussed in Chapter 4) builds structures called *path conditions* which represents the application of rules in the transformation. As these structures represent transformation executions where those rules have applied, they explicitly include the input and output elements matched over or created by those rules.

Once the set of path conditions has been created for a transformation, then pre-condition/ post-condition *contracts* are matched over these path conditions. This determines in which cases the contract is satisfied. If the post-condition does not hold on a path condition, then we have found a counter-example to the contract, and those rule combinations represented by the path condition do not satisfy the contract.

Path Conditions. An example path condition is shown in Figure 1.1, which explicitly shows the elements involved in the application of four rules in the transformation. These rules are bounded in dashed boxes and contain nodes and edges which are elements from the input and output meta-models.

The path condition represents rule application by explicitly containing the input and output elements present when the rules apply. The white top-half of the path condition is the *input graph*, while the grey bottom-half is the *output graph*. Lines between the input graph and output graph are *traceability links*, which record how the output elements were created during the execution of the transformation. This path condition can therefore be read as: "If only these four rules apply during a transformation execution, then (at least) these upper elements were present in the input model, and these lower elements are present in the output model." Section 4.1 presents more information on path conditions, including a formal definition and multiple examples.

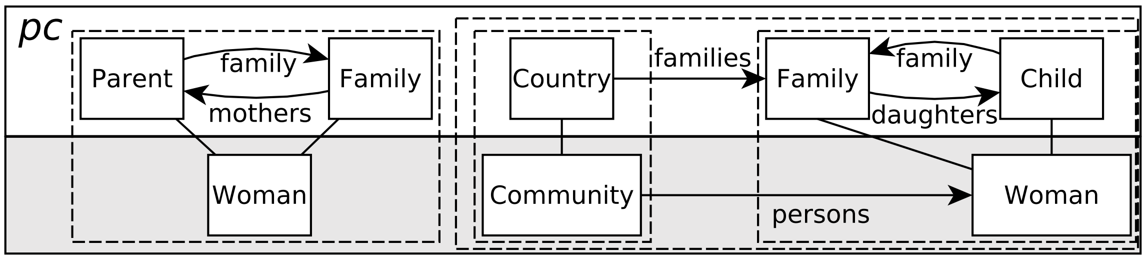


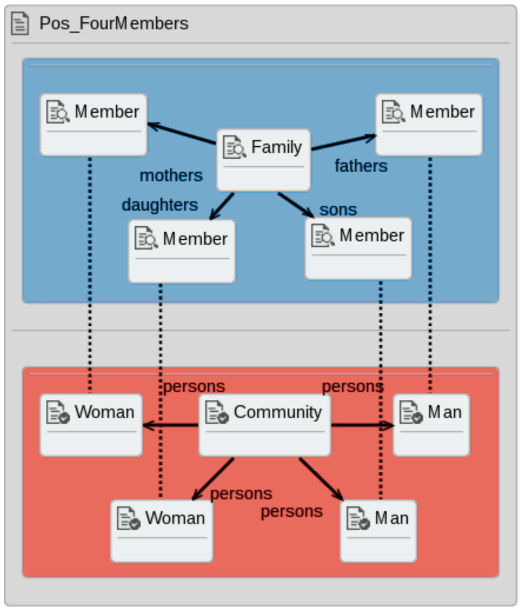
Figure 1.1 – A path condition which represents the application of four transformation rules.

Contracts. The structural pre-condition/ post-condition 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 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 (including traceability constraints).

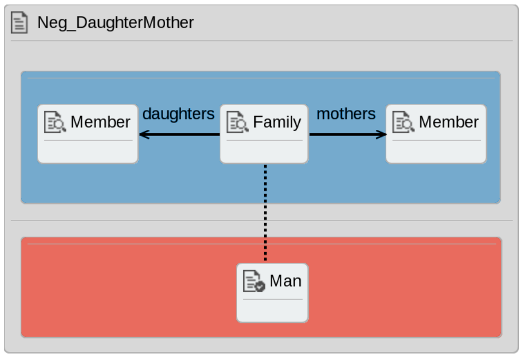
For example, the *input graph* of the contract in the upper-half of Figure 1.2a contains a *Family* element and four connected *Member* elements. The *output graph* of the contract in the lower-half of the figure contains a *Community* element, two *Man* elements, and two *Woman* elements. The dashed lines represent the restriction that the output elements must have been produced by a rule that matched over the connected input element. Note that the *Community* element is not produced directly from the *Family* element.

The contract in Figure 1.2a therefore represents the informal statement "A *Family* with a *father*, *mother*, *son* and *daughter* should always produce two *Man* and two *Woman* elements connected to a *Community*". This contract should always hold over all transformation executions of the *Families-to-Persons* transformation, which will be briefly presented in Section 2.2.1.

Note that a contract may be expected to not hold in all cases for a transformation. For example, consider the contract *Neg_DaughterMother* in Figure 1.2b, which is also from the *Families-to-Persons* transformation. The informal statement for this contract is "A *Family* with a *mother* and a *daughter* will always produce a *Man*.' As *mothers* and *daughters* should always produce *Woman* elements, then this contract should not hold in those case where only the rules involving *mothers* and *daughters* apply.



(a) *Pos_FourMembers* contract – No counter-examples found.



(b) *Neg_DaughterMother* contract – Counter-examples found.

Figure 1.2 – Contracts proved on the *Families-to-Persons* transformation.

For the *Neg_DaughterMother* contract in Figure 1.2b, our contract prover will find multiple path conditions which are counter-examples. One of these path conditions is the path condition in Figure 1.1.

Note that the pre-condition of the contract will be said to match onto the top component of the path condition (through a unique morphism), 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 verification that the transformation is working correctly, as *Man* elements have not been erroneously produced from *daughters* and *mothers* in the represented transformation executions.

If path conditions are produced as counter-examples to a contract and this was not an expected result, then this indicates an error with either the contract or the transformation. As the path conditions indicate which rule applications they represent, this allows the transformation verifier to precisely identify those rule combinations wherein an error may have occurred. This is further discussed in Chapter 4.4.

1.2.2.1 Contributions

Another contribution of this thesis is the complete definition of a contract proving approach that a) handles all constructs of the DSLTrans transformation language, including indirect links and negative elements, b) employs a novel matching technique to reduce the number of path conditions required to be built, and c) extends the abstraction relation to represent the repeated application of a rule in a path condition.

Complete Formalization. Chapter 4.4 provides the formalization of the structural contract proving process using the double-pushout approach. A notable contribution of this chapter is to handle all constructs of the DSLTrans language (such as *Any*/*Exists* elements, negative elements) which have not been addressed in past work. Thus, the contract proving approach can now verify all DSLTrans transformations.

Matching Technique. A symbolic execution approach for contract proving of DSLTrans transformations was first introduced in the thesis of Barroca [19]. This approach was then implemented as an early version of SyVOLT in Lúcio *et al.* [98].

However, this earlier approach required explicit path conditions to be produced in a combinational explosion. This is because of the 'disambiguation' approach used where elements in the path condition must be considered to overlap or not overlap with each other. An explicit path condition must be created for each possible combination of rule overlaps, which is on the order of two to the power of the number of elements in the rule. This means that the full path condition set could not be created for even small transformations, rendering the technique impractical for industrial use on larger transformations.

To avoid the creation of these explicit path conditions, a new matching morphism is detailed in this thesis. This *typed graph split morphism* has the distinction of being non-isomorphic as described in Section 2.3.2.2. This means that the morphism is able to 'split' a pattern graph over a target graph when matching, which removes the need for explicit path conditions to be produced.

For example, the path condition in Figure 1.1 is a counter-example for the contract in Figure 1.2b, as the pre-condition of the contract is said to match, but the post-condition does not. The pre-condition of the contract will only match if the split morphism is used, which is able to match the *Family* element in the contract over both *Family* elements in the path condition. This matching is presented within the contract proving context in Figure 4.20b on page 135.

Thus this thesis has the contribution of formalizing the typed graph split morphism (Section 2.3.2.2), demonstrating its use within the abstraction relation and contract proving steps (Chapter 4.4), and finally providing the algorithms used for implementation of this morphism(Section 6.2).

Repeated Rule Application. The path conditions presented in this thesis are representations of rule applications during transformation execution, as discussed in Section 4.1.3. As it is impossible to represent the repeated application of all rules within the transformation an infinite number of times, it is necessary to abstract over the number of times each rule applies.

Therefore, in this thesis we make the contribution of examining how to explicitly represent repeated application of a rule (Section 4.2.3.2), and the consequences for the abstraction relation, path condition construction and contract proving approaches. As well, we introduce a method in Section 6.4.1 which examines the contract and rules to determine if a rule must be repeatedly applied during path condition construction. This case can arise when a contract requires a rule to be applied multiple times to produce the elements matched over by the contract.

1.2.3 Prover Implementation

The culmination of our research has been the SyVOLT contract verifier, which implements the contract proving technique detailed in Chapter 4.4. This tool has already been used in transformation verification research such as the work by Azizi *et al.* [16].

1.2.3.1 Contributions

In this thesis, we make the following contributions for the SyVOLT tool: a) a description of the work-flow and algorithms employed in the contract prover as an example of multi-paradigm modelling, b) algorithmic techniques and results for reducing the time and space requirements of the contract proving approach, and c) a discussion of the contract analysis implemented in the prover, which assists the user in understanding the results of contract proof.

Work-flow and Algorithms. Our contract proving approach operates on DSLTrans transformations and rules, both of which may be represented as typed graphs and manipulated by model transformations. Thus, we have built the SyVOLT contract prover tool using a model-driven design which will be sketched in this thesis in Chapter 5.

For example, we provide a *Formalism Transformation Graph + Process Model* (Section 5.1) which explicitly details how the prover manipulates various artefacts typed by a variety of formalisms. We consider the description of the *multi-paradigm modelling* approach of our tool a contribution, as it highlights some of our successes and failures in designing a tool capable of model manipulation.

For example, in Section 5.3.3, we discuss the critical issue of creating structures which are able to match over rules. This higher-order reasoning is not widely supported by modelling tools, and we had to write the custom *PyRamify* script, which we consider to be a violation of best MDE practices. It is our hope that by highlighting the precise reasoning issues that we faced, these reasoning issues can be addressed by tool-builders in the community.

Algorithms and Techniques. The algorithms portion in Chapter 6 details the design and implementation of our matching algorithm which uses the *split morphism*. We provide pseudo-code, a brief complexity discussion, and a presentation of matching results on a benchmark.

Algorithmic techniques to reduce the time and space requirements are required during the path generation and contract proving process to ensure the verification of reasonably-sized transformations. In this thesis, we present three techniques along with complexity arguments and experiments on our set of case studies (Chapter 7). 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.

The first technique is *parallelization* where the path condition generation and contract proof processes are divided amongst different worker threads. We present in this thesis an overview of this process, along with experimental results showing how the number of threads affects the time taken for prover operation.

The act of *slicing* the transformation is our second technique. This operation chooses only those rules in the transformation which are required for a contract to be proved, dramatically reducing the time required to prove contracts. This approach is a static analysis to determine which rules need to apply (and how many times) for successful contract validation. This technique was introduced in Oakes *et al.* [112]

and expanded upon in Oakes *et al.* [113]. In this thesis, we contribute an example of slicing for a contract, a complexity discussion, and an expanded results section.

The third contribution in the area of prover algorithms is the notion of *pruning*. In this optimization, we detect required containment links in the meta-model, and remove any path conditions that do not respect those links. This reduces the number of path conditions in our symbolic execution and therefore decreases the time taken to prove contracts. In this thesis, we provide a brief overview of pruning, an example of the technique, and results showing the benefit for our case studies.

Transformation and Contract Analysis. As there may be errors during the building of the transformation or contract, the contract prover also conducts a number of static and dynamic checks which are detailed in this thesis. We detect rule and contract dependencies such that if they cannot be satisfied, this is reported before symbolic execution begins.

It can also be difficult to understand why a contract fails in the proving process. This thesis will detail in Section 6.4 a number of techniques to allow the user to understand why a contract is failing, namely which rules cause the contract to succeed or fail, and which elements the contract cannot find in a sample failed path condition.

1.3 Author Contributions

This section will discuss the precise contributions of the thesis author to each of the publications listed on page viii.

The publication of Selim *et al.* [137] applied an early version of the SyVOLT contract prover to the GM-to-AUTOSAR case study which is briefly detailed in Section 7.2. This conference paper was the result of a collaboration between McGill University and Queen's University. Levi Lúcio wrote the description of the contract proving in discussion with the author. The author then presented the paper with Gehan Selim at the International Conference on Graph Transformation (ICGT) in July 2014.

During that time, the author helped write two submissions of the technique for the Software and System Modeling journal. In particular, the author was heavily involved in discussions concerning the entire work, and was the main author on the sections of the paper explaining the path condition generation process.

The description of the semantics of DSLTrans was based on set-theory and did not address all of the constructs used in the DSLTrans language (*Any/Exists* elements, negative elements, etc.). The constructive criticism of the reviewers indicated that this formalization was not complete, and that an alternative formalization based in category theory would be more appropriate. This work was distributed as a technical paper [98], and the reformalization effort was put aside for a later time.

At the MODELS conference in 2015, the author presented a demonstration paper on the SyVOLT prover [97]. This paper was written by Levi Lúcio and the author, while the Eclipse front-end was implemented by Cláudio Gomes. As well, the paper was reviewed by our colleagues at Queen's University. Note that the chapter on the SyVOLT prover (Chapter 5) is based on this paper, but has been expanded substantially by the author.

Also presented at the MODELS 2015 conference was the conference paper on converting ATL to DSLTrans through a higher-order transformation. Javier Troya and Manuel Wimmer came up with the idea, implemented the higher-order transformation, and wrote the text about the higher-order transformation in the paper. The author performed the experiments and wrote the majority of the paper, with Levi Lúcio as second author on the SyVOLT portion.

This paper was invited to become a journal paper in 2016 for the Software and System Modeling journal [113]. During the expansion process, each contributor added detail to their section as listed in the preceding paragraph. Again, the author wrote the majority of the text about the SyVOLT prover, performed all experiments, and wrote the results section.

In 2016, Levi Lúcio led an effort to develop a front-end for SyVOLT in the Meta-Programming System (MPS), as described in Section 5.2.3. The author assisted with defining constructs and constraints for editing DSLTrans transformations and contracts. As well, the importer plug-ins such that MPS can import transformations, contracts, and meta-models were original ideas and implementations by the author. These plug-ins are also discussed in Section 5.2.3.

The author then reformulated the material in Lúcio *et al.* [98] to produce two journal-length papers. The first was submitted to the Formal Aspects of Computing journal [111] but was rejected, while the second has been published as a technical paper [110]. Note that the second paper will be submitted to a journal upon the successful publication of the first paper.

These two journal papers form the basis of Chapter 3 and Chapter 4.4. Note that the changes in material between the original paper from 2014 and the updated versions are substantial, such that very little of the original structure and formalization remain. While these papers were written in discussion with Lúcio, the author has rewritten practically all text. In other words, all additions made compared to Lúcio *et al.* [98] are the author's. These additions are described in Section 1.2.1 and Section 1.2.2.1.

In particular, the typed graph split morphism is a substantial difference to the technique presented in Lúcio *et al.* [98]. This split morphism allows for path conditions to abstract over a much larger set of input-output models for the transformation than was possible before. This decreases the cost to build these path conditions and dramatically increases the scalability of the contract prover. The split morphism was an original idea by the author. As well, all integration of the split morphism into the contract proving approach is the author's work, as well as the implementation in the SyVOLT tool (Section 6.2).

The SyVOLT prover idea and initial prototype was developed by Levi Lúcio, as well as the initial implementations of slicing and pruning. However, these implementations were redone by the author on account of the complications introduced by the split morphism. The author also implemented the parallelization optimizations and the transformation/contract analysis. Note that the analysis section forms the basis of the workshop paper on debugging of contracts [114].

All experiments and result analyses in this paper were performed by the author.

1.4 Thesis Organization

The next chapter of this paper, Chapter 2 will provide background on model transformations including and introduction to the notion of meta-modelling. An example of the DSLTrans language will follow, including selections from an example transformation. The chapter finishes with an introduction to the formal concepts used in the rest of the paper, such as typed graphs and graph morphisms. These concepts will be used to formalize the semantics of the DSLTrans language and the contract proving approach.

Following this, Chapter 3 describe the semantics of DSLTrans using the double-pushout approach. The structures of DSLTrans necessary for graph matching and rewriting are presented in Section 3.1, while the actual semantics themselves are explained in Section 3.2.

Chapter 4 discusses our verification of DSLTrans transformations. Our approach symbolically executes rules from the transformation, relying on an abstraction relation defined in Section 4.1. Through resolution of dependencies, this procedure creates a set of *path conditions*, which indicate possible input and output models for the transformation (Section 4.2).

Our path condition building algorithm builds path conditions that both *represent* and *cover* the infinite set of transformation executions, as discussed in Section 4.3.

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 4.4 with a discussion of validity presented in Section 4.5. A short discussion of the constructs available in the contract language is contained in Section 4.6 with examples from our case studies.

Chapter 5 will then provide a discussion of our multi-paradigm SyVOLT tool, which can efficiently prove structural contracts on DSLTrans transformations. The work-flow of the tool is presented in Section 5.1, while following sections introduce the two front-ends available to construct transformations and contracts, the ATL higher-order transformation, and the algorithms involved in the back-end of the prover.

Following this, Chapter 6 discusses some of the algorithms and optimization techniques employed in our contract proving tool. Section 6.2 will discuss our implementation of the split morphism which is at the core of our techniques. The algorithmic optimization techniques such as parallelization, slicing, and pruning are presented in Section 6.3, along with complexity arguments and experimental results. Finally, this chapter finishes with Section 6.4 which details the analyses we perform on transformations and contracts to detect errors and assist the user in correcting them.

Chapter 7 details the case studies used for our research. These case studies are diverse in size and complexity, and demonstrate how our technique is applicable to a variety of applications including industrial usage. The rules and contracts for each transformation are presented, along with a selection of how contracts hold or do not hold on each transformation.

In Section 8 we present work related to the DSLTrans language, the contract verification approach, and our verification tool. Finally, a detailed conclusion of the material will be presented along with avenues of future work in Chapter 9.

Chapter 2
Background

This chapter will provide the needed background for this thesis on transformation verification. First, we introduce the concepts of model transformations, which act on models to manipulate their elements. Next, we introduce the DSLTrans language and relevant constructs with the assistance of an example. Finally, Section 2.3 presents the fundamental definitions required for the formalization of DSLTrans (Chapter 3) and the contract proving approach (Section 4).

2.1 Model Transformations

The concept of *model transformations* is central to the idea of model-driven engineering. These transformations may perform a number of operations on a model, such as creating, updating, or deleting model elements. As well, transformations may be classified in a variety of ways [102], such as optimisation, normalisation, or abstraction/refinement. Transformations may also be used for the translation of a graphical model to source code, or a system might also be simulated using model transformation rules, whereby the rules continually update the model and evolve the system over time [47].

Figure 2.1 shows the components involved in a model transformation. In this diagram, a model is fed as input into a model transformation, which produces a model as output. The solid arrows represent the flow of data, while the dashed arrows represent the *conforms to* or *is typed by* relation. This relation means that the elements in the model are instantiations of the elements in the meta-model, and may be subject to constraints defined in the meta-model such as multiplicity restrictions.

The input model M_{input} conforms to the input language MM_{input}, while the output model M_{output} conforms to the output language MM_{output}. Note that the input and output languages may also be termed the *source* and *target* languages.

If MM_{input} and MM_{output} are the same language, then this is termed an *endogenous* model transformation [102]. Otherwise, the transformation is *exogenous*. As well, transformations are *in-place* if the input and output models are the same, or *out-place* if they are different. In our verification research, we focus on *exogenous* and *out-place* transformations.

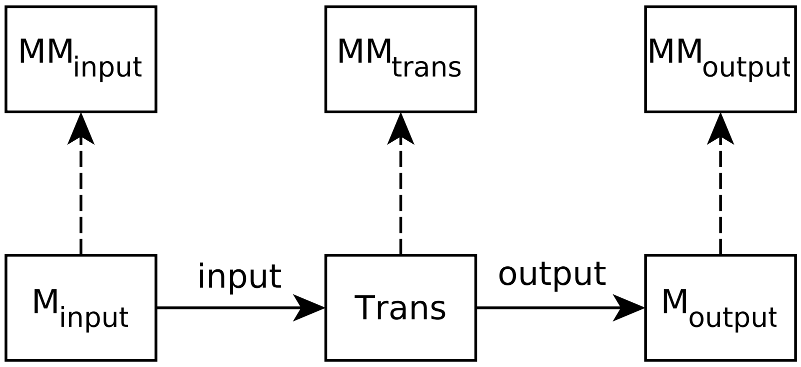


Figure 2.1 – Relationships between a model transformation and the input and output models/meta-models.

Note that a model transformation itself conforms to some language [28], marked as MM_{trans} in Figure 2.1. This transformation language contains concepts for defining and scheduling *transformation rules* which perform the actual model matching and rewriting [87]. In this thesis, the transformation language will be DSLTrans, as further explained in Section 2.2 on page 22.

2.1.1 Transformation Rules

A transformation rule is the essential component of a model transformation. In this thesis, we will focus exclusively on graph rules, which operate on typed graphs. These rules are scheduled in some order in the transformation, and they match over elements in the input model, and produce (or delete) elements in the output model.

Let r be a rule as in Figure 2.2a on the next page. r is mainly composed of two graphs, with optional negative application condition graphs as explained below. The first graph is the left-hand graph (LHS), which represents the condition for which the rule can apply. The second graph is the right-hand side (RHS), which defines the action to be performed on the graph when the LHS matches (creating, updating, or deleting elements).

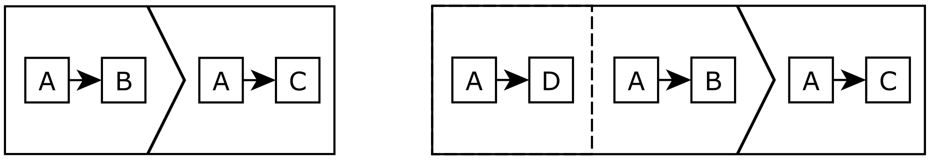
Note that both the LHS and the RHS will contain elements typed according to *pattern languages*. These pattern languages are representations which allow the rules to represent constraints and actions on the input and output languages. Pattern languages are created from the input and output meta-models through a process known as RAMification [85], which allows for the *relaxation*, *augmentation*, and *modification* of the language to form the pattern language.

An example of this process is to allow an abstract class to appear in the pattern language, which requires the relaxation of the abstract property of that element. This is further discussed in Section 5.3.3 on page 183, where we discuss our implementation of the RAMification procedure.

Rule Application. For a rule to apply, there must be a match between the LHS of the rule and the model being transformed. That is, it must be possible to match the types and structure of elements in the LHS to the input model. If the match succeeds, then the part of the input model which has been matched will then be acted upon by the RHS.

In this thesis, we will employ the double-pushout approach to graph transformation, which is explained in Section 3.2.1.1 on page 52. In this approach, elements which exist on the LHS but not the RHS will be deleted, while elements that exist on the RHS but not the LHS will be created.

An example of a rule is seen in Figure 2.2a. In this rule, if an element of type A is found connected to an element of type B, then the B element will be rewritten with an element of type C. Note that the same A element is present in both the LHS and RHS.



(a) Rule with left-hand side and right-hand side.

(b) Rule with negative application condition.

Figure 2.2 – Examples of transformation rules.

Constraints may also be added to the rule to prevent the matching of the LHS to the input model. These constraints are formulated as negative applications conditions (NAC), which use the same pattern language as the LHS. If a rule has a NAC graph, then the rule cannot apply on an input model if there is also a match of the NAC to the input model. This concept allows the transformation designer to specify cases in which the rule should not apply.

In Figure 2.2b, the NAC is found in a dashed box to the left of the LHS. It specifies that the rule may only execute if an element of type A is not connected to an element of type D.

2.2 DSLTrans Introduction

This section will present the DSLTrans transformation language, which is the target of our verification research. DSLTrans transformations operate on an input model using *rules*, which are structured into *layers*. If a rule in a layer successfully matches on the input model, then it will produce elements in the output model.

This layer-based structure enhances the understandability of DSLTrans transformations. As discussed in Section 3.2 on page 51, layers enforce the order of rule execution. As well, the semantics of DSLTrans means that transformations will always terminate and produce consistent results, as discussed in Section 3.4 on page 74.

We introduce DSLTrans through the presentation of the *Families-to-Persons* transformation. This transformation is also one of our case studies as shown in Section 7.4, where we present the full *Families-to-Persons* transformation in Figure 7.7 on page 261 and Figure 7.8 on page 262.

The meta-models and a selection of rules for this transformation are shown to help illustrate the mechanics of DSLTrans. As well, we provide a reference to the constructs available in the DSLTrans language. This section finishes with a brief discussion of the termination and confluence properties, which are crucial to the symbolic execution approach detailed in this thesis.

2.2.1 Families-to-Persons Transformation

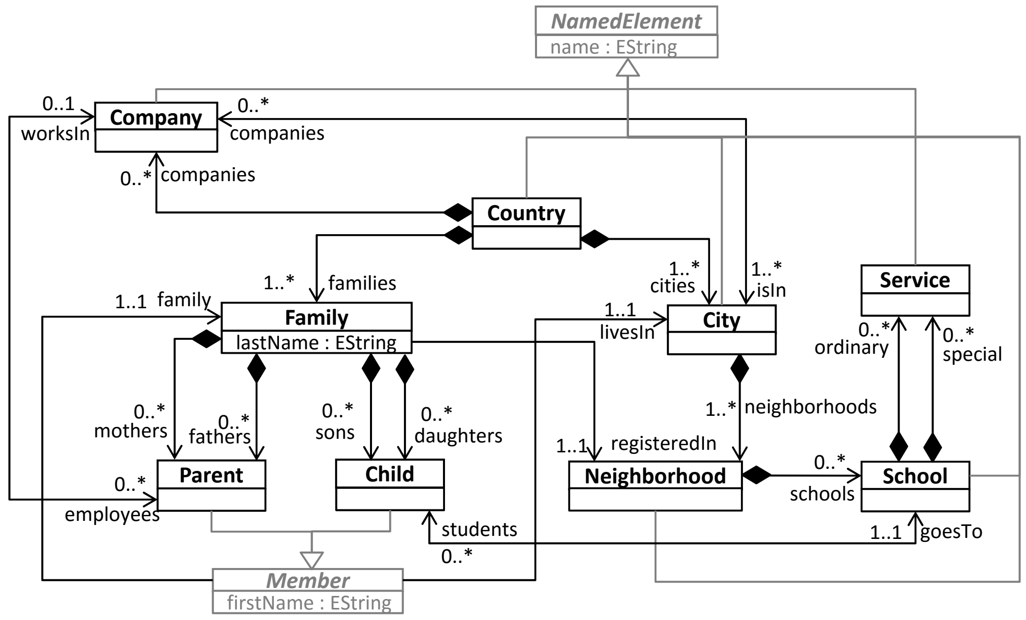
As an example DSLTrans transformation, we present the *Families-to-Persons* transformation from Oakes *et al.* [113]. This transformation takes *Families* of *Members* who live in *Countries* and *Cities*, and converts those elements into *Men* and *Women* who live in *Communities* and work at *TownHalls*.

This transformation was originally found in the ATL zoo [2], but was dramatically expanded for our work on converting ATL transformations to DSLTrans [112, 113]. Note that this conversion process is discussed in Section 5.2.1 on page 164. The *Families-to-Persons* transformation was selected as an introductory transformation as the domain is familiar to reason about. As well, the original version of this transformation has been discussed in related verification and testing research [63].

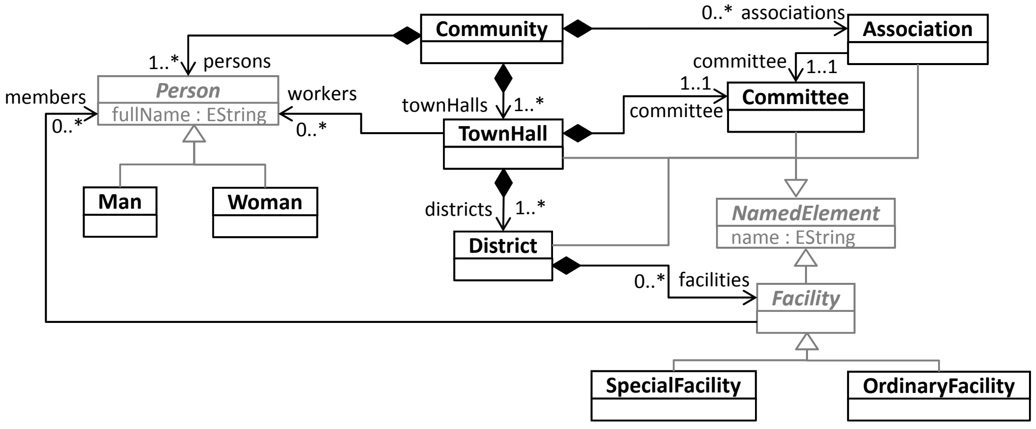
The transformation is reasonably complex enough to be representative of translation transformations. This is seen in Section 7.0.1 on page 248 with a comparison of the sizes of our case studies. As well, the *Families-to-Persons* contains interesting concepts with regards to our verification work. In particular, a number of rules produce elements in the output model which have their attributes set through manipulation of the attributes in the input model. These rules therefore test our technique's ability to correctly abstract and prove contracts on this attribute-setting.

2.2.1.1 Meta-models

The input and output meta-models of this transformation are shown in Figure 2.3, taken from Oakes *et al.* [113]. Note that abstract classes are depicted in grey and with italic names, and inheritance relationships are depicted in grey.



(a) *Families* meta-model.



(b) *Persons* meta-model.

Figure 2.3 – Meta-models for the *Families-to-Persons* transformation.

Figure 2.3a shows the input meta-model for the *Families-to-Persons* transformation. In this meta-model, the root element is a *Country*, which contains *Companies*, *Families*, and *Cities*. In each *Family*, there are *Members* where some are *Parents* who work in the *Companies* and some *Members* are *Children* who go to *School*. These *Schools* contains *Services* which may be *special* for special-needs programs, or *ordinary* for ordinary services. As well, these *Schools* are contained in *Neighborhoods* which are contained in *Cities*. *Family* elements have a *lastName*, while *Member* elements have a *firstName*, and all other elements have a *name* attribute.

The output *Persons* meta-model is shown in Figure 2.3b. In this meta-model, *Communities* contain *Persons*, *TownHalls*, and *Associations*. *Persons* are either *Men* or *Women*[1] who work at *TownHalls* or attend *Facilities*. The *Facilities* are divided into *SpecialFacilities* and *OrdinaryFacilities*. The *TownHalls* and *Associations* have a common *Committee*. *Persons* have *fullNames*, while most other elements have a *name*.

2.2.2 Rules

The essential unit of a DSLTrans transformation is the rule, containing a *MatchModel* and an *ApplyModel*. In Figure 2.4 on the following page, the *MatchModel* is the white box in the top half of the rule containing two elements, while the *ApplyModel* is the yellow box in the bottom of the rule containing one element.

The semantics of rule application, formally described in Section 3.2.1 on page 51, 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. For example, Figure 2.4 shows the *Father2Man* rule. In this rule, 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.

Note that these elements are technically patterns over the *Families* and *Persons* meta-model shown in Figure 2.3. That is, when we refer to the *Parent* element in the rule, it is the *Parent* element in the pattern language.

1. The authors note that the meta-models lack nuance with respect to gender.

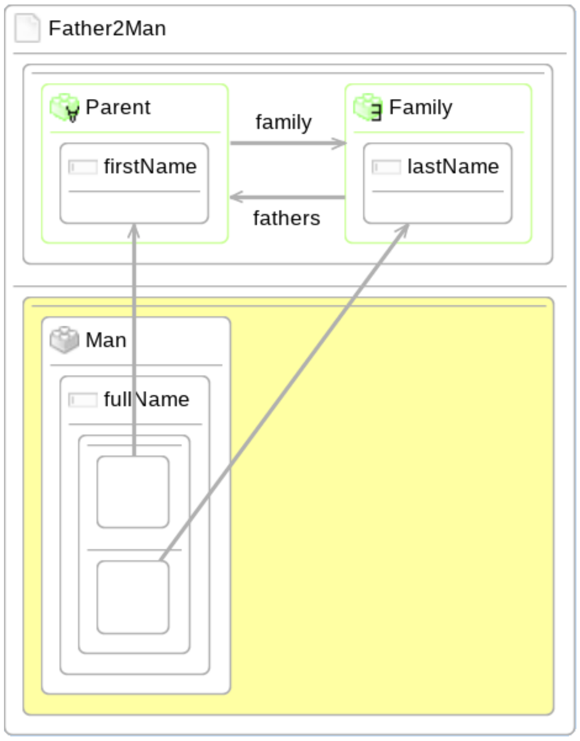


Figure 2.4 – The *Father2Man* DSLTrans rule.

Also present in this rule example is the copying of attribute values from the *MatchModel* to the *ApplyModel*. This is represented by the lines between the match elements and the apply element in Figure 2.4. 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*. This setting of attribute values is discussed in Section 3.2.3 on page 59. As well, Figure 2.4 shows the symbols to denote *Any* and *Exists* elements, as discussed later in this section.

2.2.3 Layers

The construction used for the scheduling of DSLTrans rules is a *layer*. A transformation is composed of layers where each layer contains a set of transformation rules. Layers in DSLTrans are organized sequentially such that each layer in turn will match over an input model and produce elements in the output model.

Note that while the transformation will execute layer-by-layer, rules in a layer will apply 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 produce elements (termed *out-place*). Thus, the semantics of rule execution is that all rules in the layer match as many times as possible, before they apply at once in a mass production, which is described in Section 3.2.7 on page 70. This separation of matching and rewriting steps is necessary for DSLTrans to be *confluent* by construction as discussed in Section 2.2.6.

In Figure 2.5 we present a selection of layers of the *Families-to-Persons* transformation, where each layer contains one rule. In this case, the layers will execute in sequence from left to right, as denoted by the arrows in the figure.

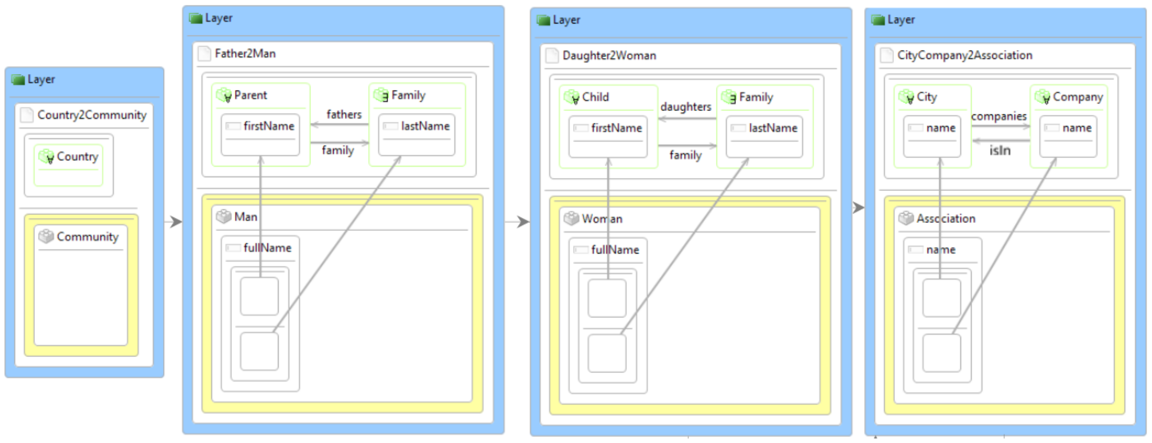


Figure 2.5 – Four layers selected from the *Families-to-Persons* DSLTrans transformation.

2.2.4 Backward Links

DSLTrans rules specify how elements of the output model were created from specific elements of the input model during rule application. 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 2.4 applies, traceability links are created from the matched *Parent* and *Family* elements to the produced *Man* elements in the output model. This building process is discussed in Section 3.1.4.2 on page 50.

The *backward link* construct in DSLTrans rules is used to match over these traceability links. Figure 2.6 depicts a rule containing two backward links. The backward links are denoted as dotted lines, connecting the *Country* and *Community* elements and the *Child* and *Woman* elements. These links enforce a dependency that the *Community* element must have been produced by a rule that matched on the *Country* element in an earlier layer of the transformation.

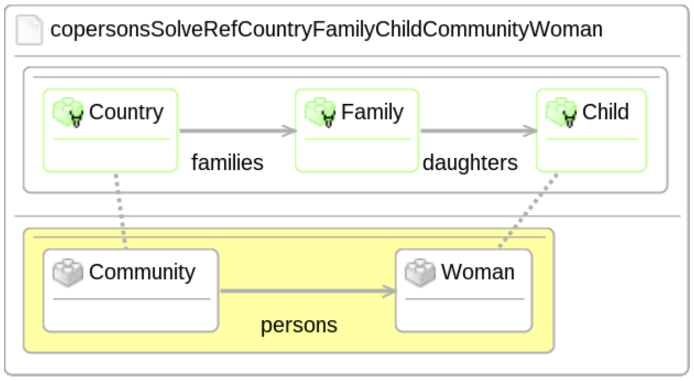


Figure 2.6 – A DSLTrans rule with backward links

These backward links are thus used during the matching process of a rule. In this process, 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* 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 2.6 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. This is formalized as the *matcher* and *rewriter* constructs for a rule in Section 3.1.4 on page 47.

2.2.5 DSLTrans Constructs

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 Section 7.4 on page 259 which presents the rules in the transformation.

Note that formal details for the syntax and semantics of these constructs are found in Sections 3.1 and 3.2. As well, extra reference material is found in the DSLTrans manual [65], which also contains instructions for executing DSLTrans transformations in the Eclipse environment.

— **Match Elements**: Match elements represent elements which will match over all elements of the corresponding type (or subtype) in the input model when the transformation is executed. Match elements include those present in the *MatchModel* of the rule, as well as those in the *ApplyModel* connected to backward links.

  Note that the RAMification process [85] (Section 5.3.3) is used to create the matching elements, such that a *School* match element can match over elements of type *School* in the target model.

— **Existential Matching**: The *Exists* construct allows selecting at most one result when a match element matches onto an input model (Section 3.2.6). These are denoted as *Exists* elements, as opposed to *Any* elements.

— **Direct Match Links**: Direct match links represent labelled associations typed by the source meta-model. 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 can exist a non-empty path without repeated elements between the matched elements for the indirect link to match. (Section 3.2.4). The typing of associations is ignored when matching indirect links.

— **Backward Links**: Backward links connect elements of the *ApplyModel* and the *MatchModel* of a DSLTrans rule to represent dependencies on element creation by previous layers of the transformation. During the execution of the transformation, backward links will match over traceability links.

These traceability links are created when a rule applies between the match elements and apply elements, as defined in Section 3.1.4.2.

— **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. The semantics for negative elements is found in Section 3.2.2.

— **Apply Elements and Apply Links**: Apply elements and apply links are similar to match elements and match links, but are instead those elements produced by the application of a rule.

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, as described in Section 3.2. In particular, apply elements that are not connected to backward links will create elements in the output model, while apply links will always be created in the output 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 Kleene star operator 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.

— *Concat*: Used to concatenate atoms, wildcards, and attribute references.

— **Match Attributes**: Uses the attribute language to create constraints on the values of attributes in the input model. Thus, the application of rules can be restricted. For example, an element could be restricted to match only when the element has a name attribute with a value of $\_\_pre$.

— **Apply Attributes**: Apply attributes can be created from concatenating *String* literals and/or references to match attributes.

2.2.6 Termination and Confluence Properties

DSLTrans has two important properties guaranteed by construction of the language: all match and rewrite operations are both *terminating* and *confluent* [21]. We require these properties for our contract verification technique as explained in Chapter 4.4.

Termination means that a DSLTrans transformation will stop executing after a finite number of steps. A transformation is confluent when, for a given input model, the outcome of executing the transformation is always the same. An examination of these properties under the double-pushout approach and with all DSLTrans constructs is found in Section 3.4 on page 74.

These properties stem from the fact that the transformation executes sequentially, one layer at a time. As well, the looping constructs (indirect links, multiple rule matches) used in DSLTrans are always finite by construction. Therefore, it is impossible to write a DSLTrans transformation which does not terminate.

DSLTrans' transformations are also strictly out-place. This means that the model being matched over cannot be modified, and that no elements can be deleted or modified in the output model. This is crucial in maintaining confluency for the execution of a transformation.

These restrictions may make writing some transformations difficult (or even impossible). For example, these restrictions rule out the use of DSLTrans for the specification of a simulation language as rules cannot infinitely fire. However, DSLTrans is sufficient for expressing *translation* model transformations [13] which translate models between source and target meta-models. Examples of translation transformations are presented as case studies in Section 7 on page 248.

2.3 Fundamental Definitions

This section introduces formal structures and concepts to explain the syntax and semantics of DSLTrans and the technique for proving contracts on DSLTrans transformations.

We begin our formalization by introducing (labelled and typed) graph concepts that will be used as mathematical building blocks. In particular we introduce our representation of graphs and the morphisms between them. This section is based on well-known concepts from graph theory, with Ehrig *et al.* [50] used as a primary reference.

2.3.1 Graphs

The (labelled) 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. We begin with a definition for a directed multi-graph (a graph allowing multiple edges between two vertices).

Definition 1. *Graph*
A graph is a 4-tuple $\langle V, E, (s,t) \rangle$ *where:*

— V *is a finite set of vertices;*

— E *is a finite set of directed edges connecting the vertices* V*;*

— V *and* E *are disjoint;*

— (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;*

— *Edges* $e \in E$ *are noted* $v \xrightarrow{e} v'$ *if* $s(e) = v$ *and* $t(e) = v'$*;*

2.3.1.1 Typed Graph

The DSLTrans transformation language is targeted at transforming domain-specific models in the model-driven engineering domain. Therefore, we must employ *typed graphs*, where the types of vertices and edges are drawn from a particular *meta-model*. These meta-models will play a part in Definition 9 on page 44, which specifies how DSLTrans rules consist of multiple typed graphs with potentially different meta-models for the match and apply components.

This typing is provided by a *type graph*, which provides the vertex and edge types to be used in a typed graph. A *typed graph* is then defined in Definition 2. where the function τ gives typing information for each vertex and edge in the graph.

Definition 2. *Typed Graph*
A typed graph is a 5-tuple $\langle V, E, (s, t), \tau \rangle$ where:

— *V, E, s, t are as in graphs;*

— *The function τ provides the mapping for each vertex and edge to a type graph;*

— *The set of all typed graphs is denoted* TG*;*

— *We define the empty graph $\varnothing$ to be the typed graph with all empty functions and sets.*

For example, the match graph for the rule in Figure 2.6 on page 29 is typed by the *Families* meta-model (presented in Section 2.2.1). The vertices in the type graph would include *Member* and *Family*, while an edge marked *fathers* would exist between those vertices in the type graph.

As well, the typing information found in the transformation domain may rely on inheritance relations. In particular, our notion of graph matching must handle sub-type information for vertices. For example, the *Member* element in a rule must match over both *Parent* and *Child* elements, due to the inheritance relationship in the *Families* meta-model. Note that edge sub-typing is not considered.

We construct a partial ordering $\leq$ on the vertex types in a type graph to be used during the matching of graphs and structures. For the treatment of inheritance during matching, we rely on the work of *de Lara et al.* [46], where vertex types are replaced with all sub-types during the matching process as required using *clan morphisms*. Thus, inheritance considerations are handled transparently.

Note that for simplification purposes, we will not represent edge cardinalities or containment relationships given by a meta-model in our notion of type graph. In fact, we require these conditions to be relaxed (as in the RAMification procedure [85]) to perform our graph rewriting, as incomplete output graphs will be produced during the execution of the transformation.

2.3.1.2 Sub-graph

A typed sub-graph 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 9).

Definition 3. *Typed Sub-graph*
Let $\langle V, E, (s,t), \tau\rangle = g$, *and* $\langle V', E', (s',t'), \tau'\rangle = g'$, *where* $g, g' \in \mathrm{TG}$.

g' *is a typed sub-graph of* g, *written* $g' \sqsubseteq g$, *iff:*

— $V' \subseteq V$;

— $E' \subseteq E$;

— $s' = s|_{E'}$;

— $t' = t|_{E'}$;

— $\tau' \subseteq \tau$

2.3.2 Graph Morphisms

The formal development of our technique requires the definition of relations between typed graphs that preserve (some) graph structure and the types of vertices and edges, 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*.

2.3.2.1 Typed Graph Morphism

The first typed graph morphism we define is standard in the graph-matching literature, where it is used to find an isomorphic match of the structure of the pattern graph in the target graph [43]. This morphism 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 $\langle V, E, (s,t), \tau \rangle = g$, *and* $\langle V', E', (s',t'), \tau' \rangle = g'$, *where* $g, g' \in \mathrm{TG}$.

A typed graph morphism $f : g \to g'$ *is a function* $f : (f_v, f_e)$ *such that:*

— $f_v : V \to V'$;

— $f_e : E \to E'$;

— $\forall v_1 \xrightarrow{e} v_2 \in E$:

— *Let* $v'_1 = f_v(v_1), v'_2 = f_v(v_2), e' = f_e(e)$;

— $v'_1, v'_2 \in V', e' \in E'$;

— $s'(e') = v'_1$, $t'(e') = v'_2$;

— $\tau(v_1) \leq \tau'(v'_1), \tau(v_2) \leq \tau'(v'_2), \tau(e) \leq \tau'(e')$

This last condition handles matching of types, such that the vertex and edge types in the pattern are the same or super-classes of the types in the target graph. That is, the types of the vertices (or edges) are obtained with τ or τ' and then the $\leq$ partial ordering determines whether the types match or not. An equivalent formulation is to consider the use of *clan morphisms*, which replaces elements with their sub-types in the pattern as required during matching.

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

An *injective* typed graph morphism f from g onto g' is written $f : g \overset{inj}{\to} g'$. A *surjective* typed graph morphism from g onto g' is written $g \overset{surj}{\to} g'$.

2.3.2.2 Typed Graph Split Morphism

The improvements in this thesis to the abstraction relation (Section 4.1) and path condition creation technique (Section 4.2) 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.

Definition 5. *Typed Graph Split Morphism*
The typed graph split morphism exists between the pattern graph g and the target graph g' when a third interface graph h exists that satisfies the following properties:

— $h \in \mathrm{TG}$

— $\exists p : (h_V \overset{surj}{\rightarrow} g'_V, h_E \overset{inj}{\rightarrow} g'_E)$

— *This is a surjection on the vertices, but an injection on the edges*

— $\exists q : h \overset{inj}{\rightarrow} g'$

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

This *split morphism* is primarily different to Definition 4 in the matching of vertices. In the split morphism, 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 2.7. Note that on the right-hand side, the graph h is found as a sub-graph isomorphism within the target graph. On the left-hand side, the vertices are allowed to overlap in the non-injective morphism between h and the pattern graph. However, the edges in h must be found injectively in the pattern graph g.

This unusual morphism allows our technique to 'split' our pattern graph vertices to match over multiple vertices in the target graph. Note that the mapping from a pattern vertex to the (set of) target vertices is created by composing the p and q morphisms. That is, for an element a, the mapping *m(a)* produces a set such that

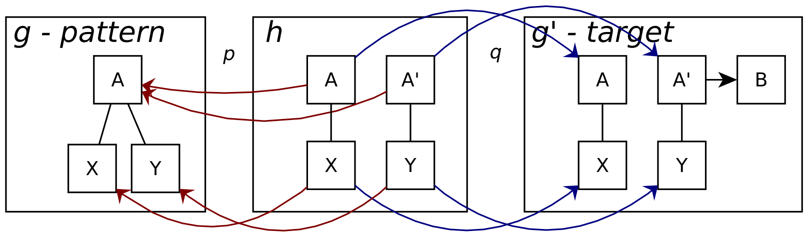


Figure 2.7 – Constructing the interface graph h for the existence of the typed graph split morphism.

the morphism p gives the element a, and the morphism q provides the elements of the set. $m(a) = \{q(n_1), q(n_2), \ldots q(n_n) \mid p(n_i) = a\}$.

The ability for this morphism to allow overlapping is critical to our approach as our target graphs represent a structure that suffers from vertex duplication. That is, two vertices of a particular type in the target graph may only represent one underlying element in an abstracted structure.

A concrete example for when this 'one-to-many' matching is required is found in Section 4.4.2.3 on page 133. Briefly, the issue is that a contract needs to be matched onto a *path condition* structure representing multiple rules (Section 4.1.3). Therefore, one element in the contract may match onto multiple elements in different rule components. This morphism is our way of unifying elements in different rules without the combinatorial explosion of generating all unifications of the rules explicitly, as is performed by Barroca [19].

Note that this split morphism is not equivalent to a non-injective morphism. In a non-injective morphism, *multiple* pattern nodes may match on *one* target node. In contrast, the split morphism allows *one* pattern node to match on *multiple* target nodes.

2.3.2.3 Transitive Closure

DSLTrans matching constructs allow for the matching of *indirect links* where rules may match over indirect paths between elements.

The definition of transitive closure in Definition 6 explicitly creates all transitive edges for that graph. This explicit creation allows us to use indirect links in a straightforward way during the matching of the rule, as discussed in Section 3.2.4 on page 61.

Note that we will enforce a restriction that transitive edges are only created between vertices on paths with no cycles.

Definition 6. *Transitive Closure*
Let g be a typed graph $\langle V, E, (s,t), \tau \rangle \in \mathrm{TG}$. Then we define the transitive closure $g^* = \langle V, E', (s', t'), \tau' \rangle \in \mathrm{TG}$ to be:

— $E_{tc} = \{v_1 \xrightarrow{e_{tc}} v_2 \mid e_{tc} \notin E$ *and a path exists in V between v_1 and v_2 without cycles*$\}$

— $E' = E \cup E_{tc}$

— $s'(e_{tc}) = v_1$, $t'(e_{tc}) = v_2$;

— $\tau(e_{tc}) = indirect$

2.3.2.4 Links Between Graphs

In the developments that follow, we create structures which consist of two typed graphs, with *links* between them. For example, these structures may be a DSLTrans rule (Definition 9 on page 44) or an input-output model (Definition 12 on page 47).

Definition 7 describes how the links between the two typed graphs is simply a set of edges, grouped with the source and target functions. However, it is important to note that the source and target elements of each link are in the typed graphs. Therefore, the links are connecting the two typed graphs.

Definition 7. *Link Homomorphism*
Let a link be a tuple of edges and the source / target functions. That is, $l = \langle E, (s,t) \rangle$.

Definition 8 defines a morphism to match *links*. An example for its use would be to match the backward link construct for DSLTrans rules, as discussed in Section 2.2.4 on page 28. This link morphism is necessary such that the backward links from a DSLTrans rule (Definition 9 on page 44) can match onto a input-output model containing traceability links (Definition 12 on page 47).

Definition 8. *Link Homomorphism*
Consider two groups of links $A = \langle E_A, (s_A, t_A)\rangle$*, and* $B = \langle E_B, (s_B, t_B)\rangle$.

We define an injective morphism function $f : A \to B$ *to match* A *over* B*, such that* $\forall e_a \in E_A$*, there exists* $e_b \in E_B \mid \big(f(e_a) = e_b\big)$.

As well, the constraints (types, attributes) must match for the source and target nodes of the e_a and e_b links.

2.3.2.5 Morphisms Between Structures

The final morphism required is the mapping between the structures which are used throughout this thesis to represent transformation rules, intermediate states of the transformation, and the contracts to prove. In general, each of these structures is composed of two typed graphs, connected by links. For example, the contract structure defined in Definition 24 on page 128 is composed of a pre-condition, post-condition, and backward links between them. This structure must match onto a path condition (Definition 19 on page 86), which is composed of an input graph, output graph, and traceability links.

We create a morphism $m : S_1 \to S_2$ 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 $\langle E, (s, t)\rangle$.

This overall morphism is composed of the sub-morphisms $f : A \to A'$, $g : B \to B'$, $h : L \to L'$, where f and g are the morphisms to be found, and h is the link homomorphism described in Definition 8. 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 composed as $m = (f, g, h)$, where the sub-morphism employed is chosen based on the component of interest.

Consistency. As well, we require these morphisms to have consistency in their vertex mapping for the matching morphism m to be valid.

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

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. This is stated in Equation 2.1.

$$h(l) = l' \implies \big(g(s(l)) = s'(l') \wedge f(t(l)) = t'(l')\big) \tag{2.1}$$

Chapter 3
DSLTrans Formalization

This chapter will discuss a key contribution of this thesis: the complete and precise definition of the semantics for all DSLTrans constructs in the double-pushout approach.

This answers our first research question: *How can the semantics of a model transformation language with reduced expressiveness (DSLTrans) be precisely formalized and brought to the state-of-the-art?*

Section 3.1 begins this chapter by defining the core structures involved in DSLTrans as typed structures. Structures defined include the rule, layer, transformation, the matchers and rewriters used for the application of the rule, as well as the input-output model construct acted upon by rules.

Section 3.2 then employs these structures in a precise description of the semantics of DSLTrans in the double-pushout approach (Section 3.2.1.1). The semantics of the rules, layers, and the overall transformation will be presented, as well as an examination of all constructs such as *Exists* elements, indirect links, and negative elements. These latter constructs have not been precisely defined in earlier works on DSLTrans, and thus a contribution of this thesis is to describe them and present examples of their use.

A discussion of the complexity of DSLTrans execution follows in Section 3.3 to provide a sense of the scalability of DSLTrans.

Finally, the proof sketches for confluence and termination for DSLTrans are presented in Section 3.4. These proof sketches originated in the thesis of Barroca [19], and have been updated to account for the new formalization and new constructs (*Exists* elements, indirect links, negative elements) addressed in this thesis.

3.1 DSLTrans Structures

This section will detail the constructs involved in a DSLTrans transformation such as rules, layers, and the matchers and rewriting used in rule application.

Note that in this section, we define structures which are very similar in composition, with each containing two typed graphs, as well as a link component. This approach of typed graph composition is preferable as the similarity between structures aids comprehension of our proving technique. In particular, in the path condition generation and contract proving portions of this thesis (Section 4.2 and Section 4.4), we employ multiple morphisms between the various components of these constructions.

We also note that this three-part structure approach directly reflects the implementation in the DSLTrans transformation engine and contract prover tool.

These structures can also be represented as one large typed graph. In this case, elements would be annotated by the component they originate from, and Definition 3 on page 35 (Typed Subgraph) would be used to select an appropriate projection.

3.1.1 Transformation Rule

Recall from Section 2.2 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 2.4 in Section 2.2. These match and apply graphs are similar [1] to the *left-hand side* (LHS) and *right-hand side* (RHS) (Section 2.1)

1. The difference is that elements in the *Apply* graph connected to backward links are matched in DSLTrans rules (Section 2.2.4).

found in the model transformation literature. When the elements from the rule's LHS are found in the input model, then the RHS elements are produced in the output model.

Definition 9. *Transformation Rule*
A DSLTrans transformation rule is a six-tuple $\left\langle Match,\ Apply,\ backward,\ V_{Neg},\ E_{Neg},\ V_{Exists},\ E_{indirect}\right\rangle$*, where:*

— $Match, Apply \in \mathrm{TG}$;

— $Match \cap Apply = \emptyset \wedge Apply \neq \emptyset$.

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}.

— $backward = \langle E_{back}, (s_{back}, t_{back})\rangle$;

— E_{back} *contains the backward links;*

  — $E_{back} \cap E_{Match} = \emptyset$, $E_{back} \cap E_{Apply} = \emptyset$.

  — $\forall e \in E_{back} : \tau(e) = back$

— (s_{back}, t_{back}) *is a pair of functions*
$s_{back} : E_{back} \rightarrow V_{Apply}$ *and* $t_{back} : E_{back} \rightarrow V_{Match}$ *that respectively provide the pattern and target vertices for each backward link*

  — *Note the source of backward links is a vertex in* V_{Apply} *while the target is a vertex in* V_{Match}

— $V_{Neg} \in V_{Match}$ *is the set of negative vertices in the match graph;*

— $E_{Neg} \in E_{Match} \cup E_{back}$ *is the set of negative edges in either the match graph or the backward links;*

— $V_{Exists} \in V_{Match}$ *is the set of Exists elements in the match graph;*

— $E_{Indirect} \in E_{Match}$ *is the set of indirect links.*

RULES *is the set of all rules.*

A transformation rule also includes backward links (and potentially negative backward links) to define dependencies between rules, as introduced in Section 2.2.4.

These backward links are placed in a separate component in the formalization of the DSLTrans rule.

A rule may also contain negative elements in its match graph, such as match elements or direct and indirect match links. These elements are held in the set V_{Neg} and E_{Neg}. These elements will be considered during the matching procedure as formally described in Section 3.2.

Match graphs can also include indirect links that are used to match over the transitive closure of the associations in a model. These indirect links are held in the set $E_{indirect}$. An apply graph does not include indirect links as they are used only for the matching of elements.

Finally, match elements in DSLTrans rules can be denoted as *Any* or *Exists* elements, as explained in Section 3.2.6. These *Exists* elements are held in the set V_{Exists}.

We additionally impose that for the well-formedness of a DSLTrans rule, there always exists an element or edge to be created in the *Apply* graph to be created. That is, either there are edges present in the *Apply* graph, or there is a vertex in the *Apply* graph that is not the source of a backward link. This is presented in Equation 3.1.

$$E_{Apply} \neq \emptyset \quad \vee \left(\exists v \in V_{Apply}, \forall e \in E_{back} : s_{back}(e) \neq v\right) \tag{3.1}$$

Rule Well-Formedness

3.1.2 Layer and Transformation

Definition 10 and Definition 11 formalize a DSLTrans transformation, which is composed of a sequence of layers where each layer is composed of a set of rules.

Definition 10. *Layer*
A layer is a finite set of transformation rules: $\{r_0, r_1, \ldots, r_n \mid r_i \in \textsc{Rules}\}$.

The set of all layers is denoted LAYERS.

Note that the order of the rules within the layer does not matter, due to the semantics of DSLTrans rule execution. These semantics ensure that rules are independent and cannot act on the output of other rules within the same layer as discussed in Section 2.2.6 and Section 3.4.1,

Definition 11. *Transformation*
A DSLTrans transformation is a three-tuple, consisting of a finite sequence of layers, along with the type graphs for the input and output meta-models. $\langle (l_0, l_1, \ldots, l_n \mid l_i \in \textsc{Layers}), mm_{in}, mm_{out} \rangle$.

The set of all transformations is denoted as TRANSFORMS.

Note that the order of layers in a transformation is important, as this ordering defines the execution of the transformation.

The type graphs for the input and output meta-models are used to type the typed graphs composing the rules and intermediate states within the transformation.

3.1.3 Input-Output Model

Before describing the semantics of a DSLTrans model transformation in Section 3.2, 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.

Note that the structure of a input-output model is intentionally very similar to a DSLTrans rule to support the matching and rewriting of elements.

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 apply. 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 3.2.1.1 on page 53.

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

- $Input, Output \in \mathrm{TG}$*;*
- $|Input| \geq 0, |Output| \geq 0$*,* $Input \cap Output = \emptyset$*;*
- $trace = \langle E_{trace}, (s_{trace}, t_{trace}) \rangle$
  - E_{trace} *contains the traceability links;*
  - $E_{trace} \cap E_{Input} = \emptyset$*,* $E_{trace} \cap E_{Output} = \emptyset$*;*
  - $\forall e \in E_{trace} : \tau(e) = trace$
  - (s_{trace}, t_{trace}) *is a pair of functions* $s_{trace} : E_{trace} \to V_{Output}$ *and* $t_{trace} : E_{trace} \to V_{Input}$ *that respectively provide the source and target vertices for each traceability link.*

The set of all input-output models for a transformation tr is denoted IOM_{tr}.

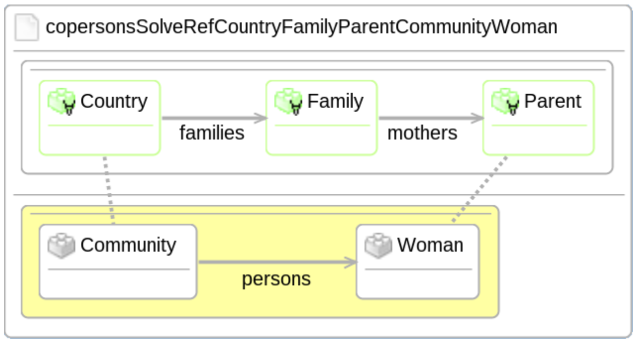
Note that as this structure represents an execution of the transformation, the type graph for the input and output graphs will be sourced from those contained in the transformation (Definition 11).

We define a utility function *getTransformationIOM*: *IOM* $\to$ *Transforms*. This (trivial) function returns the transformation that the input-output model was built for from the set of all transformations. 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 application of rules in that transformation.

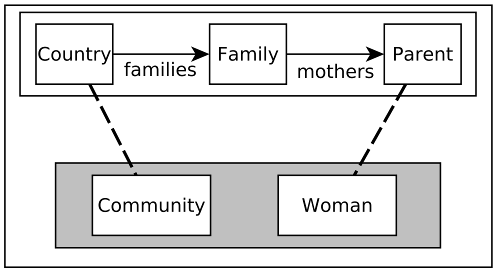
3.1.4 Matcher and Rewriter

This section will discuss how the *matcher* and *rewriter* structures for a DSLTrans rule are constructed. These structures are required for the application of rules in the double-pushout approach, as they act as the rule's left-hand side and right-hand side as mentioned in Section 2.1 on page 19.

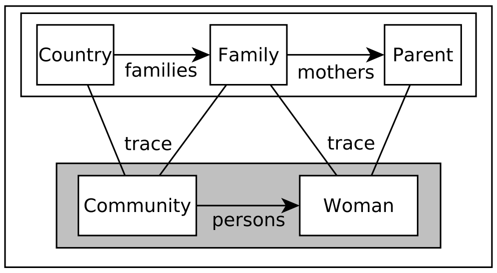
An example of matcher and rewriter construction from a rule is presented in Figure 3.1. A rule from the *Families-to-Persons* transformation is shown in Figure 3.1a. This rule was selected as it contains both backward links and performs edge creation in the apply graph. The matcher for this rule is shown in Figure 3.1b, while Figure 3.1c shows the rewriter.



(a) The *copersons...* rule.



(b) The matcher for the rule.



(c) The rewriter for the rule.

Figure 3.1 – The matcher and rewriter structures created for a rule.

Note the presence of the *persons* edge in the rewriter. As this edge is not present in the matcher, it will be created in the output model when this rule applies.

3.1.4.1 Matcher of a Transformation Rule

A DSLTrans rule can only apply if the elements present in the match graph appear in the graph being matched over. As well, the backward links in the rule are additional constraints on how the rule can match. Therefore, the matcher for a rule 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 may also be negative elements and links present in the match graph, as well as negative backward links. These elements will be extracted into a *negative application condition* (NAC) during rule matching, as explained in Section 3.2.2 on page 56.

Definition 13. *Matcher of a Transformation Rule*
Let the transformation rule $r = \left\langle Match^r, Apply^r, backward^r, V^r_{Neg}, E^r_{Neg}, V^r_{Exists}, E^r_{indirect}\right\rangle$.
We define r's matcher, noted $\lceil r \rceil$*, to be a seven-tuple* $\left\langle Match^m, Apply^m, backward^m, V^m_{Neg}, E^m_{Neg}, V^m_{Exists}, E^m_{indirect}\right\rangle$*, where:*

- $Match^m \in \mathrm{TG}, Apply^m \in \mathrm{TG},\ backward^m = \langle E, (s,t)\rangle$
- $V^m_{Match} = V^r_{Match}$;
- $E^m_{Match} = E^r_{Match}$;
- $V^m_{Apply} = \{s_{back}(e) \mid e \in E_{backward}\}$;
- $E^m_{Apply} = \emptyset$;
- $(s^m_{Match}, t^m_{Match}, \tau^m_{Match}) = (s^r_{Match}, t^r_{Match}, \tau^r_{Match})$;
- $(s^m_{Apply}, t^m_{Apply}, \tau^m_{Apply}) = (s^r_{Apply}, t^r_{Apply}, \tau^r_{Apply})$;
- $backward^m = backward^r$;
- $V^m_{Neg} = V^r_{Neg} \mid_{V^m}$ *is the set of negative vertices;*
- $E^m_{Neg} = E^r_{Neg} \mid_{E^m}$ *is the set of negative edges;*
- $V^m_{Exists} = V^r_{Exists} \mid_{V^m}$ *is the set of Exists elements;*
- $E^m_{indirect} = E^r_{indirect} \mid_{E^m}$ *is the set of indirect links.*

In Figure 3.1b, the matcher of the rule contains the *Country*, *Family*, and *Child* elements from the match graph of the rule, as well as the backward links connected to the *Community* and *Woman* elements. Therefore for the rule to apply on a target graph, all of these elements must be present and the backward links must successfully match over traceability links.

Note that the white and grey boxes in the matcher are used to highlight the similarity in structure to the rule itself. The white box contains elements from the source meta-model of the rule, while elements from the target meta-model of the rule are contained within the grey box.

3.1.4.2 Rewriter of a Transformation Rule

For the double-pushout approach, we must also construct the rewriting (or replacement) graph which will produce additional elements or attributes in the graph being rewritten. Note that in other transformation languages, the rewriter may define a deletion of elements – those that appear in the matcher but not the rewriter – but DSLTrans only permits the addition of elements.

The construction of this rewriter is essentially the same as the underlying rule. However, new traceability links will be created between most match and apply vertices to indicate the proper traceability for element creation. As well, we discard the negative elements in the rule, as they are used only for matching.

Note that the backward links present in the matcher are still present as traceability links in the rewriter. This is purely a change in visual syntax, and the underlying link is not modified.

Definition 14. *Rewriter of a Transformation Rule*
Let the transformation rule $r = \big\langle Match,\ Apply,\ backward,\ V_{Neg},\ E_{Neg},\ V_{Exists},\ E_{indirect}\big\rangle$.
We define r's rewriter, noted $\lfloor r \rfloor$*, to be a three-tuple* $\big\langle Input,\ Output,\ trace\big\rangle$*, where:*

— $Input \in \mathrm{TG},\ Output \in \mathrm{TG},\ trace = \langle E, (s,t)\rangle$

— $V_{Input} = \{m \in V_{Match} \mid m \notin V_{Neg}\}$;

— $E_{Input} = \{e \in E_{Match} \mid e \notin E_{Neg}\}$;

— $(s_{Input}, t_{Input}, \tau_{Input}) = (s_{Match}, t_{Match}, \tau_{Match})$;

— $Output = Apply$;

— $trace = \forall (a \xrightarrow{e} m) \in trace$:
$m \in V_{Match} \mid m \notin V_{Neg},\ a \in V_{Apply},\ s_{trace}(e) = a,\ t_{trace}(e) = m.$

The set *trace* will contain the traceability links between elements matched in the application of this rule, and the elements being created. Note that traceability links will only be created from non-negative elements, and their direction is from the output elements to the input elements, similar to backward links.

As well, the backward links which were present in the rule are represented as traceability links in the rewriter. This is to represent the fact that the links are not deleted by the rewriter. For example, in Figure 3.1c the backward links in the matcher are still present in the rewriter, but instead as denoted as traceability links.

Therefore in the rewriter, only the two traceability links coming from the *Family* element will be created, along with the *persons* association in the apply part of the rule. All other elements and links will be retained.

3.2 DSLTrans Semantics

This section will discuss the semantics of DSLTrans transformation execution, by grounding the language in the elegant *double-pushout* approach to graph transformation. This approach, detailed by Ehrig *et al.* [50], employs morphisms to embed the matching and rewriting processes of the rule within the target graph.

We start our explanation of transformation execution by describing the application of a DSLTrans rule on an input model to produce elements in the output model. Then, we explain how layers and transformations are executed, in a bottom-up approach.

3.2.1 Application of a DSLTrans Rule

The application of a DSLTrans rule is a two step process. First, the elements and links contained in the rule's matcher are found in the target graph. Then, the elements and links contained in the rule's rewriter are produced in the target graph.

Note that for simplicity the following explanation of rule application will be presented twice. The first and simpler case is where the DSLTrans rule does not contain the indirect link construct. This case is diagrammed in Figure 3.2. Then, the explanation will be extended in Section 3.2.4 on page 61 to build the intermediate graphs necessary for the indirect links to match.

3.2.1.1 Double-Pushout Diagram

The core idea of the double-pushout approach to graph rewriting is to present graph matching and rewriting in terms of morphisms. These morphisms allow us to compartmentalize the actions of finding a match of the pattern graph in the target graph, and the addition (or deletion) of elements and edges [50].

To frame this in terms of *category theory*, the objects are structures of typed graphs, and the arrows between the objects are morphisms between the structures. Thus, all operations are within the category of typed graphs and graph morphisms. For instance, performing the application of a rule will involve matching all three components of a rule matcher (Definition 13 on page 49) onto all three components of an input-output model (Definition 12 on page 47). A concrete example of this matching process is displayed in Figure 3.3 on page 54.

The top half of Figure 3.2 shows a *production* $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$, where L, K, and R are finite typed graphs respectively representing the left-hand side of a rule, the gluing (or interface) graph, and the right-hand side. Note that K is termed the *interface graph* of L and R because K contains the elements common to both L and R.

l and r are graph morphisms *embedding* the elements of K within L and R respectively.

The application of a rule depends on the production p and a graph G, and whether there exists a mapping m between L and G. If so, then the rest of the double-pushout diagram can be built in the following way as succinctly stated by Syriani [143]:

1. Find a mapping of L onto G. This mapping is denoted $M = m(L)$.
2. Remove $L - K$ (the elements to be deleted) from M such that the gluing condition $(G - M) \cup k(K) = D$ still holds.
3. Glue $R - K$ (the elements to be created) to D in order to obtain H.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

Figure 3.2 – A double-pushout diagram

If the above steps can be accomplished, then the boxes labelled (1) and (2) in Figure 3.2 are *pushouts*. That is, the morphisms will commute on either side, as further discussed on page 55:

$$m\big(l(K)\big) = G = f\big(k(K)\big) \text{ and } g\big(k(K)\big) = H = n\big(r(K)\big)$$

Rule Application

We shall now interpret Figure 3.2 in terms of the matching and rewriting process for a DSLTrans rule. Let us consider a production for a rule $r = \langle \mathit{Match}, \mathit{Apply}, \mathit{back} \rangle$, with its matcher $\lceil r \rceil$ and rewriter $\lfloor r \rfloor$.

For this first explanation, we constrain our rules to not include any indirect links, and to have no negative elements or links in the match component or the backward link component, as in the equations below. The indirect and backward link constructs are examined separately in Section 3.2.1.1 on page 56.

$$V_{Neg} = \emptyset \tag{3.2}$$

$$\nexists e \in E_{Match} \mid e \in E_{Neg} \vee e \in E_{indirect} \tag{3.3}$$

Requirements for the First Double-Pushout Explanation

An illustrative example of double-pushout graph transformation is presented in Figure 3.3. In this example, the rule from Figure 3.1 on page 48 is applied to an input-output model through the matching of the Matcher and Rewriter structures. Note that the input-output model records the application of previous rules, as the

Man element in the apply portion of the target graph is connected to the *Community* element.

Note that this figure represents each three-component structure (Matcher, Rewriter, input-output model) as a unified graph. This is to aid comprehension and bring our approach closer to the double-pushout literature. As the components in the structures are disjoint, this can be done without loss of generality.

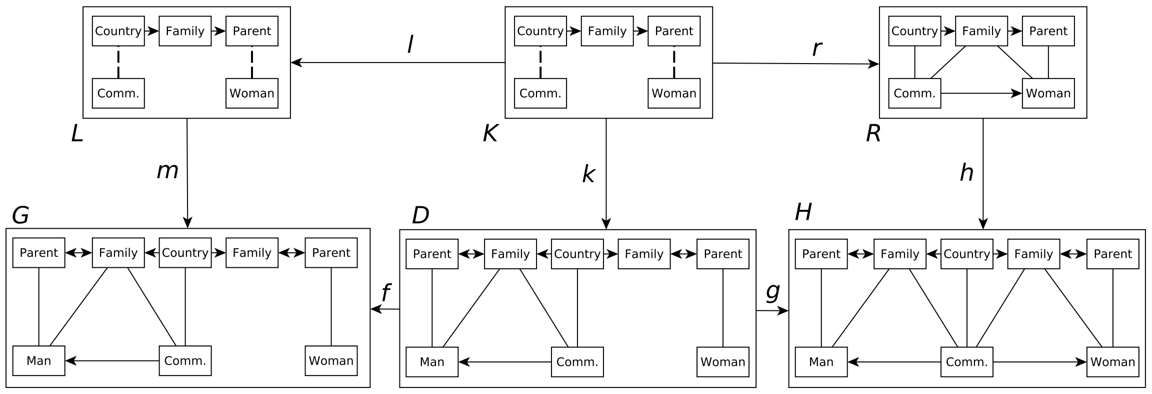


Figure 3.3 – An example of double-pushout graph transformation for the *Families-to-Persons* transformation.

The components in Figure 3.3 of interest to model transformation are:

— L: The matcher $\lceil r \rceil$ of the rule

— R: The rewriter $\lfloor r \rfloor$ of the rule

— G: The graph to be matched/rewritten

— H: The rewritten graph

Matching Components. G is the target graph which is to be matched on. In the semantics of DSLTrans, this G will be an input-output model $iom = \langle Input, Output, trace \rangle$. m is the typed graph morphism between the matcher of a rule r, denoted $\lceil r \rceil$ and the components of the input-output model.

Note that the explanation for the presence of negative elements in $\lceil r \rceil$ is discussed in Section 3.2.1.1 on page 56, as this requires further explanation.

Interface Components. Recall that K is the interface graph between L and R. That is, elements in K appear in both L and R. Elements which are to be deleted by the production appear in L but not K, and elements to be created are present in R but not K. In the absence of indirect links, we note that K is isomorphic to L as elements cannot be deleted in DSLTrans rules and therefore the morphism l is an isomorphism.

The morphism r from K to R is an injective typed graph morphism. Note that as described in Section 2.3.2.5 on page 40, this morphism will be composed of sub-morphisms between the *Match* $\rightarrow$ *Input*, *Apply* $\rightarrow$ *Output*, and *backward* $\rightarrow$ *trace* components. As such, consistency must be maintained, as described in Equation 2.1 on page 41.

D is the interface graph between the original graph G, and the rewritten graph H. Similar to K, D contains elements from the intersection of G and H.

Again, as DSLTrans rules do not delete elements, D will be isomorphic to G in this case, and the morphism f will be an isomorphism. The typed graph morphism k will embed the K interface graph within the interface graph D.

Rewriting Components. The typed graph morphism r gives the embedding of the interface graph K into the rewriter of the DSLTrans rule R. The elements present in R but not K indicate new elements to be produced by the rule.

Finally, the typed graph morphism g embeds the interface graph D into the rewritten graph H, providing the location for the addition of the new elements from R. Note that the r and g morphisms will be isomorphisms from K (resp. D) to a subgraph of R (resp. H).

Commutativity. For the double-pushout approach to be valid, it must be shown that the two squares in Figure 3.2 commute.

The commutativity of these morphisms on the left-hand side is given by the lack of deletion of elements in DSLTrans. As explained above, this means that the L and K graph, and the G and D graphs are isomorphisms of each other. The isomorphism morphisms l and f will therefore commute with the m and k morphisms, whether m and k are the typed graph morphism or the typed graph split morphism. For the right-hand side, an equivalent argument can be made using the fact that the morphisms running horizontally are sub-graph isomorphisms, which will commute with the morphisms running vertically.

Finally, we note that the existence of the *Exists* elements may affect the commutativity of these morphisms. As described in Section 3.2.6 on page 65, this is due to the way that the *Exists* elements are restricted in their matching on the target graph. Note that this restriction is non-deterministic, which may also affect confluence as discussed in Section 3.4.1 on page 74. Therefore, to ensure commutativity, we require that the *Exists* elements are consistently matched to the same elements in the target graph, as is also required for confluence in Section 3.4.1.

Additional Constructs

The following sections will extend the above double-pushout explanation to include additional DSLTrans constructs such as negative application conditions, attributes in the rule, and indirect links. Note that while the former two constructs are mentioned in the thesis of Barroca [19], they were not precisely defined.

DSLTrans match elements can also be divided into *Any* and *Exists* types. This construct is separately explained in Section 3.2.6 on page 65, as these types only affect the matching of the rule when the rule is repeatedly applied to the target graph.

3.2.2 Negative Application Conditions

DSLTrans rules can contain negative elements and links to prevent rule application, reflecting similar constructs in other transformation languages such as AGG [147]. The components needed for matching are the *positive application condition* (PAC) for what pattern is to be found in the target model, as well as a *negative application condition* (NAC) which prohibits rule application if it is found. In DSLTrans, these two conditions are intertwined in the rule matcher construct.

To place this in the double-pushout approach, we follow the approach of Habel *et al.* [70] to represent the matching of the NAC and the PAC as two different morphisms on the target graph. These PAC and NAC graphs are extracted from the matcher by using relatively trivial morphisms.

3.2.2.1 PAC and NAC Construction

The PAC is created by selecting only the positive vertices and edges from the rule matcher. However, to prevent the creation of dangling edges where an edge is not connected to two vertices, we only extract positive edges which are between two positive vertices.

Similarly, the NAC is created by extracting the negative vertices and edges. As well, to avoid dangling edges in this case, we also include any positive vertices connected to those negative edges.

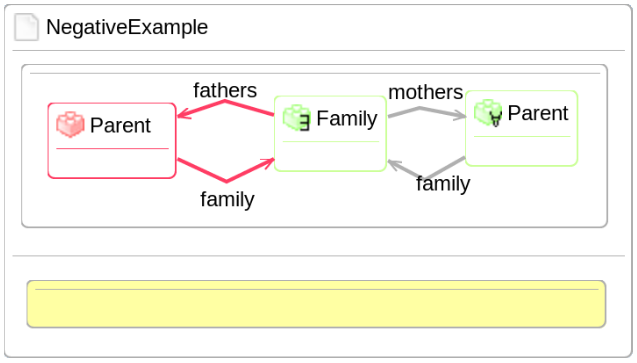
Let this extraction process be defined using the morphisms a and b, which embed the NAC and PAC in the matcher L, as in Figure 3.4. These morphisms rely on the set V_{Neg} and E_{Neg} defined for the rule's matcher in Definition 13 on page 49.

$$\begin{tikzcd}
NAC \arrow[r, hook, "a"] \arrow[dr, "n", "/" marking] & L & PAC \arrow[l, hook', "b"'] \arrow[dl, "p"'] \\
& G &
\end{tikzcd}$$

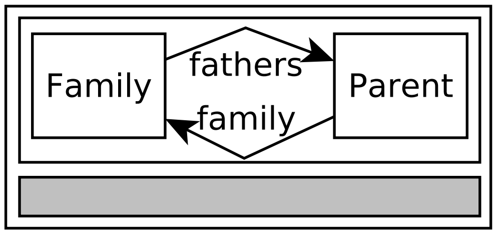
Figure 3.4 – Extraction of the PAC and NAC graphs.

An example of this extraction process is seen in Figure 3.5. The rule in Figure 3.5a contains a negative *Member* element connected to a positive *Family* element through a negative association *father* in the match graph. Thus, this rule should match over families that contain a *mother*, but do not contain a *father*. Note that the apply graph is purposefully left empty as this example focuses on the matching portion.

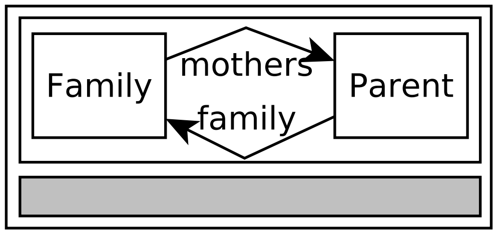
The positive application condition (PAC) graph in Figure 3.5c shows that the positive elements are extracted, consisting of the *Family* and the *Parent* connected by



(a) An example rule with negative elements.



(b) The extracted negative application condition (NAC).



(c) The extracted positive application condition (PAC).

Figure 3.5 – Creating the PAC and NAC for a rule with negative elements.

the positive *mothers* and *family* associations. For the negative application condition (NAC) graph in Figure 3.5b, the negative *Parent* element is extracted, along with the negative *fathers* and *family* associations. The positive *Family* element is also extracted to the NAC to prevent dangling edges with no connected vertices.

3.2.2.2 Matching of PAC and NAC

To place this within the double-pushout approach as seen in Figure 3.2, we declare that for L to match onto G, then p from PAC to G must exist, and n from NAC to G must not exist. These new morphisms are displayed in Figure 3.4 on the previous page.

Note that these morphisms can be easily composed with the double-pushout diagram in Figure 3.2 by replacing $L \xrightarrow{m} G$ in that figure with Figure 3.4. Thus, whenever we use the graph matching morphisms $\rightarrow$ and $\rightarrowtail$ in this thesis, we also include the matching of the positive elements, and preclude the matching of the negative elements.

3.2.3 Attributes

Another available construct in DSLTrans rules is to match over or store attributes on elements. For example, a *Person* element could have an *firstName* attribute of type *String* and a value of *"James"*.

DSLTrans currently considers only *String* attributes due to their ease of implementation. However, the attribute language could be easily extended to other attribute types. For example, element attributes could contain integer values, and rules could have constructs for matching ranges of values.

Also note that due to the semantics of DSLTrans, attributes cannot be removed or modified for an element once that element is created.

Attributed graphs can be addressed in the double-pushout approach in a number of ways [116]. In this section, we discuss one approach in the literature, and briefly outline why DSLTrans cannot use this approach.

3.2.3.1 Attributes as Vertices Approach

In Ehrig *et al.* [50], attributes are represented as distinct attribute vertices, connected to the elements using node and edge attribute edges. This form of *typed attributed graph* is defined in Definition 15.

Definition 15. *Typed Attributed Graph*
A typed attributed graph is a 7-tuple
$\langle V_G, V_D, E_G, E_{NA}, E_{EA}, (s_j, t_j)_{j \in \{G,NA,EA\}}, \tau \rangle$ *where:*

— V_G *and* E_G *are the sets of vertices and edges for the graph;*

— V_D *is a finite set of data vertices;*

— (s_G, t_G) *is a pair of functions* $s_G : E_G \rightarrow V_G$ *and* $t_G : E_G \rightarrow V_G$ *that respectively provide the source and target vertices for each edge in the graph;*

— (s_{NA}, t_{NA}) *is a pair of functions* $s_{NA} : E_{NA} \to V_G$ *and* $t_{NA} : E_{NA} \to V_D$ *to connect graph vertices to attribute vertices;*

— (s_{EA}, t_{EA}) *is a pair of functions* $s_{EA} : E_{EA} \to E_G$ *and* $t_{EA} : E_{EA} \to V_D$ *to connect graph edges to attribute vertices.*

An advantage of the attribute-as-vertices approach is that normal isomorphic matching requires relatively little extra treatment [50]. If the attribute vertices are connected to a pattern vertex, then there must be an isomorphic copy on the target vertex for the match to succeed.

However, certain DSLTrans constructs (detailed in Section 2.2.5) render the approach of modelling attributes as vertices to be infeasible.

The *Wildcard* concept allows matching of arbitrary text on a match attribute. For example, it is possible to specify in a matcher that the *name* attribute for a *City* element must start with *"San"* or *"Los"*. The existence of this construct implies that a purely graph-based approach is not flexible enough, and that there must exist a resolver which is able to handle wildcards.

As well, to construct the concrete attribute values for elements in the rewriter part of the rule, the *Concat* construct can combine *Atoms* and *Match Attribute References*.

For example, the *firstName* of a person could be concatenated with a *lastName* string to form a *Man*'s *fullName*, as in the *Families-to-Persons* rule in Figure 2.4 on page 27. This concatenation is also difficult to represent with a purely graph-based approach.

3.2.3.2 Resolver Approach

In our approach, we assume the existence of a resolver for graph matching. This resolver must be able to handle the *Wildcard* construct when matching over a graph, and return a true or false result whether the wildcard matches. As well, the resolver must correctly handle the *Concat* construct, such that instead of *Concat* nodes, the concatenated *String* values are correctly created in the *output models* produced by the transformation.

During rule rewriting, new equations must be defined for nodes to define their attributes. In this case, equations on an attribute must be checked for consistency, such that only one value is given to a node's attribute. This is also performed by our simple resolver.

Therefore, the matching and rewriting of the attributes in DSLTrans must be handled by a conceptual layer above the double-pushout approach.

Our implementation of this resolver in our contract proving tool is discussed in Section 6.2.2. As well, we briefly discuss in that section our approach to storing attributes in DSLTrans rules. Attributes are stored as equations in the rule component itself, and are tuples consisting of an element identifier, attribute name, and attribute values.

3.2.4 Indirect Links

If the DSLTrans rule contains indirect links, we further modify the double-pushout described in Section 3.2.1 to handle the transitive closure that the indirect link construct implies. The main change is to build the transitive edges, match on these links, and then not carry the transitive edges forward to the final graph.

Note that this explanation is an extension of Section 3.2.1, as the explanation covers both cases where rules do or do not contain indirect links.

The component G' in Figure 3.6 represents the transitive closure of the *Input* graph of G. Recall that this transitive closure, defined in Definition 6, will add edges between vertices that are connected by any path in the graph. Note that this will lead to two nodes already connected in the starting graph to be connected again by the new edges. The isomorphism i then gives the embedding of G into a sub-graph of G'.

The matching morphism m will try to map the indirect links from L onto the newly created edges in G'. Note that the type of the newly created edges will be *indirect*, which can be matched by the indirect links in the match graph.

Once matching has occurred, all indirect links should not be present in the rewritten graph. In the rule, K will not contain the indirect links. Note that this means that the morphism l' will not be an isomorphism between L and K. The graph R and the morphism r remain the same. Correspondingly, the interface graph D does not contain the edges matched over in G' and thus the morphism f' is not an isomorphism between D and G'.

Finally, the edges created in the transitive closure should not remain in the final graph H'. This is accomplished by defining a morphism u that embeds all nodes and original edges from the graph H into the graph H'. That is, G, G', and H have a type graph that includes *indirect*, while H' does not.

3.2.4.1 Example

Figure 3.6 shows an example of the double-pushout approach to graph rewriting when indirect links are present in the rule.

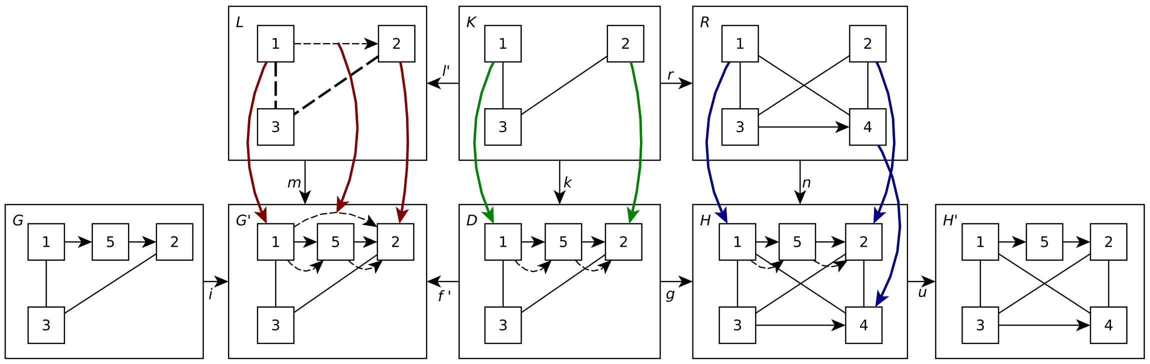


Figure 3.6 – Double-pushout example with indirect links.

Note that the *G* and *D* graph both match onto the newly created graph *G'*, by an isomorphism onto a sub-graph of *G'* that does not contain the newly created edges from the transitive closure.

The satisfaction of the indirect link in *L* is given by matching on the newly created link in *G'*. However, the indirect link in *L* does not appear in *K*. The binding sites of the *1* and *2* elements are correctly found, allowing the addition of the *4* element in the right-hand side of the rule.

As well, the graph H contains the edges added by the transitive closure, between the *1*, *5*, and *2* elements. These are removed in a final morphism u, which transfers the graph from a type graph including the indirect links to one without.

3.2.5 Repeated Application of a DSLTrans Rule

The explanation in Section 3.2.1.1 (or Section 3.2.4) is for a pair of the match and rewrite steps of a DSLTrans rule. This match / rewrite pair is called a *production* in the double-pushout literature. In this section, we define the semantics of DSLTrans when a rule will apply multiple times over an input graph, once for every set of elements where the matcher matches.

For example, let there exist an input-output model *iom*, a rule r, and a production p that takes *iom* and produces a new input-output model iom'. As the next application of the rule r, the second production p' will match on iom', and produce iom'', and so on.

However, we define that the matching morphism for the second production p' must only match on those elements present in the original input-output model *iom*. This is done to enforce the semantics that rules must not match on newly-created elements in the same layer.

To neatly express these limitations in repeated rule execution, the double-pushout approach allows us to gather all the matches on *iom*, and apply all the productions at once, known as *parallel graph productions*. We refer to Definition 3.22 (parallel graph production and transformation) in Ehrig *et al.* [50] for the creation process of this mass production p^*, as detailed in the next section.

3.2.5.1 Mass Production Creation

Section 3.2.1.1 on page 53 describes the components L, K, and R which compose a production for a rule. To create a *mass production*, which is a union of rule productions, we must first define the + operator in Equation (3.4) to combine two productions:

$$p_i + p_j = (L_i \cup L_j \xleftarrow{l_i \cup l_j} K_i \cup K_j \xrightarrow{r_i \cup r_j} R_i \cup R_j) \tag{3.4}$$

Production Combination

The + operator defined in Equation 3.4 operates on two productions and performs a disjoint union of the graphs involved. The combined production is then available to match and rewrite on the target graph, thus combining the match and rewriting steps from two productions.

$$\begin{tikzcd}
p_1 : L_1 \arrow[d, "in_L^1"] & K_1 \arrow[l, "l_1"'] \arrow[r, "r_1"] \arrow[drr, "in_K^1"] & R_1 \arrow[drrrr, "in_R^1"] & \dots & p_k : L_k \arrow[dllll, "in_L^k"'] & K_k \arrow[l, "l_k"'] \arrow[r, "r_k"] \arrow[dll, "in_K^k"] & R_k \arrow[d, "in_R^k"] \\
L = L_1 + \ldots + L_k & & & K = K_1 + \ldots + K_k \arrow[lll, "l"'] \arrow[rrr, "r"] & & & R = R_1 + \ldots + R_k
\end{tikzcd}$$

Figure 3.7 – Union of multiple productions, redrawn from [50].

As well, for convenience we define the operator $\uplus$ in Equation 3.5 to be the + operator repeatedly applied on a set of productions. Figure 3.7 shows a union of multiple productions graphically.

$$\uplus(p_1, p_2, \ldots, p_n) = p_1 + p_2 + \ldots + p_n \tag{3.5}$$

Repeated Production Combination

Rule Mass Production. To represent all applications of a rule r on a target graph g, we will create a *rule mass production* p_r^*. This production will aggregate multiple copies of the rule's production to combine all the individual match / rewrite steps into one production.

In Equation 3.6, the rule's production is repeatedly combined with itself as many times as possible, such that the final mass production still matches over the target graph g. Note that this restriction ensures that the mass production is finite, as the target graph to be matched on is also finite.

$$p_r^* = \max_{i<N} \ \uplus \, p_r \quad | \quad p_r^* \rightarrow g \tag{3.6}$$

Rule Mass Production

The negative application condition (NAC) of the rule is still relevant to the creation of this mass production. If the NAC prevents the rule from applying, then the production created will be empty. If the NAC does not prevent rule application, then it can be safely discarded,

This amalgamation of rule productions is possible because DSLTrans rules do not delete any elements and can therefore be applied at the same time. These rules are *parallel independent*, and thus p^* can be created as a union of the L, K, and R components of each production [50].

Definition 16 is the core of DSLTrans' semantics in which a production is applied to an input-output model iom_{pre} to produce another input-output model iom_{post}.

Definition 16. *DSLTrans Rule Application*

The application of a DSLTrans rule is defined as a function:

applyRule : IOM × Rule → IOM

Let $rule = \{r_0, r_1, \ldots, r_n\} \in \textsc{Rules}$*, and* iom_{pre}*,* $iom_{post} \in$ *IOMs.*
iom_{post} *will be the result of applying the rule mass production* p_r^* *to* iom_{pre}*.*

3.2.6 Any versus Exists Elements

When discussing rule productions, we must also explain constructs that affect how rules can match on the target graph. DSLTrans match elements can be classified as either *Any* or *Exists* elements. An *Exists* element provides *existential matching* where the matching location is restricted, while an *Any* element will match over all possible locations [65].

3.2.6.1 Exists Example

Consider the model in Figure 3.8, which contains *Manager* and *Employee* elements where a *Manager* element may *manage* many *Employees*.

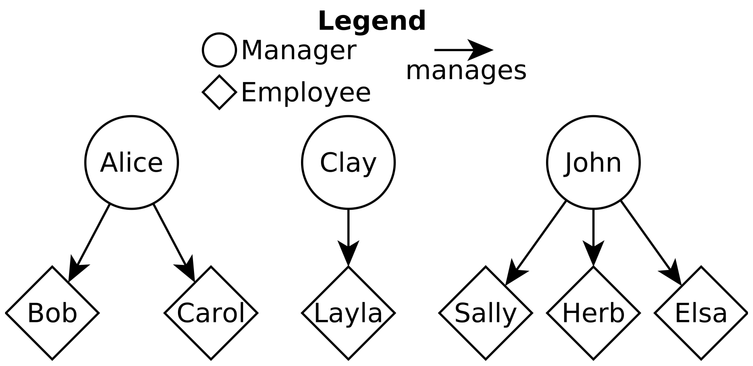


Figure 3.8 – Example model for explaining *Exists* element semantics.

A transformation designer may wish to have rules to represent such statements as *"For every pair of Managers and Employees..."* or *"For one Employee for every Manager..."*. Table 3.1 shows how the *Any* and *Exists* elements can be used in the matchers of rules to match over these different concepts.

The first column shows the elements in the matcher of the rule (using the symbols in the legend of Figure 3.8), the second column provides a statement describing the intention of the matcher, and the third column presents the results when the rule is matched over the model in Figure 3.8. Note that the × symbol represents the Cartesian Product, and is employed for conciseness.

For instance, the third matcher in Table 3.1 matches over all pairs of *Managers* and exactly one *Employee*. Note that there are two elements in the left column which are unconnected by an association, and that the *Manager* element is denoted *Any* while the *Employee* element is denoted *Exists*. This *Exists* element will be bound to exactly one element in the model. In the results column on the right, the matching locations are the pairs *Alice-Bob*, *Clay-Bob*, *John-Bob*.

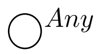
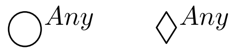
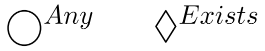
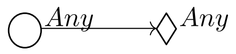
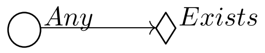
Matcher	Rule Intention	Matching Locations
	Match all managers	$\mathit{Alice},\quad \mathit{Clay},\quad \mathit{John}$
	Match all manager-employee pairs	$\{\mathit{Alice},\quad \mathit{Clay},\quad \mathit{John}\}\times$ $\{\mathit{Bob},\quad \mathit{Carol},\quad \mathit{Layla},$ $\mathit{Sally},\quad \mathit{Herb},\quad \mathit{Elsa}\}$
	Match all managers and one employee	$\{\mathit{Alice}, \mathit{Clay}, \mathit{John}\} \times \{\mathit{Bob}^*\}$
	Match all managers and their employees	$\mathit{Alice} - \mathit{Bob},\quad \mathit{Alice} - \mathit{Carol},$ $\mathit{Clay} - \mathit{Layla},\quad \mathit{John} - \mathit{Sally},$ $\mathit{John} - \mathit{Herb},\quad \mathit{John} - \mathit{Elsa}$
	Match all managers and one of their employees	$\mathit{Alice} - \mathit{Carol}^*,\quad \mathit{Clay} - \mathit{Layla},$ $\mathit{John} - \mathit{Sally}^*$

Table 3.1 – Examples of matches onto Figure 3.8. Asterisks represent deterministic selection.

Note that the selection of the *Bob* element is marked with an asterisk, as this depends on the deterministic selection as discussed in the next section.

Another use of the *Exists* element is shown in the last row of Table 3.1. This matcher is designed to match over all pairs of *Managers* and exactly one of their connected *Employees* as the *Employee* element in the left column is denoted as *Exists*. The results column on the right records that only one of each of the *Manager's Employees* are matched, again based on deterministic selection.

3.2.6.2 Enforcing Semantics

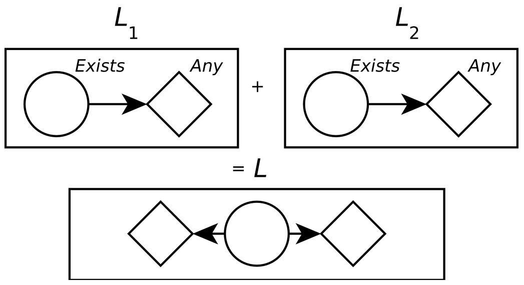
Based on the preceding description, the *Exists* element is restricted in where and how many times it may match in the target graph. These semantics are enforced during the creation of the production for the rule, such that those elements are forced to combine together or to stay separated.

Specifically, this enforcement is performed during the creation of a rule mass production (Equation 3.6), where the productions of rule are repeatedly combined together. The *Exists* construct defines how the elements in the matcher of the productions (the L graph) are combined together to specify how the production will match over the target graph.

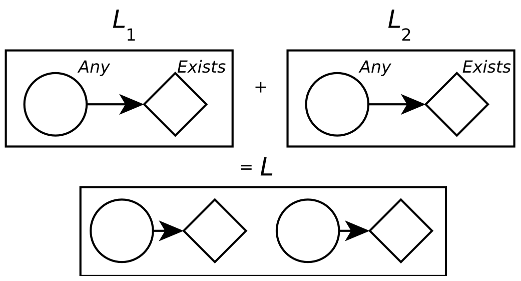
If an element is denoted as *Exists* in a rule by its presence in the set V_{Exists} in the matcher, then we specify that whenever there are multiple productions combined using Equation 3.4, that there are two restrictions on this combination.

First, if the *Exists* element is not connected by an association, or is at the tail of an association, then all copies of that element are unified amongst all copies of that matcher. This unification is demonstrated graphically in Figure 3.9a, where the matchers for two productions representing the statement *"For all employees of one manager..."* are combined.

The *Manager* element is marked as *Exists* and is at the tail of an association. Therefore, after the + operator is applied to L_1 and L_2, only one copy of the *Manager* element exists in the produced production L. Thus, this element is forced to match over only one element in the target graph.



(a) Combining *Exists* elements for unified matching.



(b) Forbidding *Exists* elements from repeated matching.

Figure 3.9 – Enforcing the semantics of the *Exists* element when combining productions.

In the second restriction, if the *Exists* element is at the head of an association, then it is the *association* that is restricted in its matching, and neither connected element can be combined. For example, the matchers being combined in Figure 3.9b represent the statement *"For one employee for all managers..."*. Therefore, both the *Manager* and *Employee* elements must match over different elements in the target graph, and cannot be unified in any way.

Figure 3.9b therefore shows the only combination of these elements, where neither the *Manager* nor *Employee* elements have been unified. This forces these elements to match onto different elements in the target graph.

If both elements were marked as *Exists* elements, then the first restriction takes precedence. Therefore, both elements and the association will be unified, such that only a single *Manager* and *Employee* pair will be matched in the target graph.

Therefore, by overloading the union operator defined in Equation 3.4, the semantics of the *Exists* element can be enforced when generating the mass production for a rule. This elegantly compacts the semantics of this construct into the double-pushout approach. However, the determinism of the construct must be examined as discussed in the next section.

3.2.6.3 Deterministic Matching

Recall the confluence property of DSLTrans (Section 2.2) which states that two transformation executions on the same input model produce the same output model. This may be impacted by the *Exists* element, which enforces a choice of which element to match over, and thus may potentially change between transformation executions. For instance, Table 3.1 shows how *Manager* elements in the matcher were matched over a single element in the target graph.

Therefore, to ensure repeatability and confluence, we stipulate that the choice of matched element be deterministic for a particular DSLTrans implementation, such that the matching locations for *Exists* elements are repeatable. This is critical to ensure confluence as discussed in Section 3.4.1.

As a possible implementation, element order could be imposed by creating comparable hashes for each element based on element connections, data, and order of creation. The matching selection would be essentially random to the user, satisfying the intended behaviour of this construct while still maintaining determinism and thereby confluence. However, note that a significant time or memory cost may be required to create these hashes depending on the specific implementation.

3.2.7 DSLTrans Layer Execution

This chapter has so far defined single and multiple application of a particular DSLTrans rule on a target graph. As well, we have provided a discussion on semantics for the constructs defined within rules, such as indirect links and negative elements. The next definitions uses these previous definitions to examine how DSLTrans layers and transformations apply rules to an input-output model

First, we define in Equation 3.7 the *layer mass production* p_l^* for a layer l. This definition combines each individual rule mass production $p_{r_i}^*$ for the rules r_i in the layer l with the union operator.

In informal terms, we are collecting all the *Match* and *Apply* graphs for each rule in the layer, with each rule represented as many times as it can match on our input model and taking into account the presence of *Exists* elements. Thus, this production will represent the combined match / rewrite steps for all the rules in the layer.

$$p_l^* = \uplus\, p_{r_i}^* \quad \forall r_i \in l \tag{3.7}$$

Layer Mass Production

Definition 17 then defines how a layer of rules in a transformation is applied to an input-output model. We define the application of rules in a layer by first creating the layer mass production as in Equation 3.7, and then applying this production to the input-output model.

Note that this will enforce the desired semantics where rules belonging to the same layer are forced to execute independently by not being able to match on the output model of the other rules. This is opposed to other model transformation approaches based on graph rewriting, where the result of each rule rewrite may be immediately usable by all other rules.

Definition 17. *DSLTrans Layer Execution*

The execution of a DSLTrans layer is defined as a function:

$applyLayer : IOM \times Layer \rightarrow IOM$

Let $layer = \{l_0, l_1, \ldots, l_n\} \in \textsc{Layers}$, *and* iom_{pre}, $iom_{post} \in IOMs$. iom_{post} *will be the result of applying the layer mass production* p^*_{layer} *to* iom_{pre}.

3.2.8 DSLTrans Transformation Execution

Our final definition is for the semantics of execution for a DSLTrans transformation. Essentially, it is the chaining of layer executions applied on an input-output model. However, we begin by discussing the conditions for executing a model transformation on an input model.

Input Conditions. Recall that an *input-output model* is the three-tuple $\langle Input, Output, trace \rangle$ (Definition 12 on page 47). Let iom_{pre} be the input-output model given as input to the transformation, and iom_{post} be the input-output model produced by the transformation.

Note that iom_{post} will contain the same *Input* graph as iom_{pre}, but will have extra elements in *Output* and *trace* as produced by the execution of the transformation.

As described in Definition 18, we create iom_{post} by repeatedly applying the productions produced by each layer in the transformation in turn. The input-output model that is the output of executing a given layer is passed onto the next layer as input. The final input-output model created will be the result of the model transformation.

Definition 18. *Execution of a DSLTrans Transformation*
The execution of a model transformation is a function consisting of a chaining of executions of those layers $l_n \in \textsc{Layers}$ *within the transformation, and having the signature* $applyTransformation : IOM \times Transforms \rightarrow IOM$.

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

$iom_{post} = applyLayer\big(\ldots\ applyLayer(applyLayer(iom_{pre} \times l_0) \times l_1) \ldots \times l_n\big)$

Note that this approach makes it possible to 'chain' transformations. In this case, the input-output model produced by one transformation would be passed as input to another transformation, assuming that the typing information of the transformations and input-output model are compatible.

3.3 Complexity of DSLTrans

This section will present a brief discussion of the complexity of DSLTrans transformation execution to provide a rough sense of the scalability of the DSLTrans language. The complexities will be discussed following the presentation order of the semantics.

Note that the complexities discussed here are based on the T-Core model transformation engine [145]. T-Core provides a number of model transformation primitives (*Matcher*, *Rewriter*) that have previously been used to implement model transformation languages including DSLTrans [91]. Further details on the T-Core engine can be found in Section 5.3.2.

Matching. A crucial component in a graph transformation engine is the efficient matching of pattern graphs onto target graphs, which is the well-known NP-complete *sub-graph isomorphism problem.*

The work of Cordella *et al.* [43] introduces the VF2 algorithm to perform this matching, which has a time and spatial complexity for graph V of $\Theta(|V|^2)$ and $\Theta(|V|)$ in the best case, and $\Theta(|V|!\,|V|)$ and $\Theta(|V|)$ in the worst case.

Note that this algorithm also separates matching each node into *syntactic* and *semantic* feasibility steps. The *syntactic* step checks whether the pattern node and the target node have compatible structures.The *semantic* step ensures that typing and attribute matching is considered, which means that the complexity depends on the resolution algorithm. For example, the complexity of matching and rewriting attributes depends on the number and type of those attributes, and the complexity of the possible operations.

In Section 3.6, we discuss how repeated rule application is represented in the double-pushout approach by creating a *mass production*. To create this mass production, a rule's production is repeatedly combined with itself as many times as will still match over the target graph. As well, the semantics of the *Any* and *Exists* elements must be taken into account.

However, an equivalent formulation of this mass production creation is to consider finding all possible matches of the pattern graph on the target graph. According to Provost [122], the worst-case complexity for the VF2-like algorithm in that work is $\mathcal{O}(n^m)$, where $n = |V_{Matcher}|$ and $m = |V_{Target}|$. This case occurs when every pair of vertices are compatible.

Note also if negative application condition graphs (NACs) are present in a rule, these also have to be matched onto the target graph to see if the rule is disallowed, as in Section 3.2.2. This may increase the number of matches that need to be performed.

Indirect Links. One addition to the complexity of the matching procedure for DSLTrans is the process by which indirect links are matched. As discussed in Section 3.2.4, if indirect links exist in the pattern graph, a transitive closure is first performed on the target graph before matching proceeds.

Using the Floyd-Warshall algorithm to discover all possible paths in the target graph, the complexity of this operation is $\mathcal{O}\big(\,|V|^3\,\big)$ [171].

Rewriting. Once the matching step is complete, then the rule's rewriter can be embedded into the target graph, and the new elements can be added. This step is therefore based on the size of both the matcher and the rewriter [145].

As each node and edge in the pattern graph is joined onto the target graph, the complexity of this step is $\mathcal{O}\big(\,|V_{pattern}| + |E_{pattern}| + |V_{target}| + |E_{target}|\,\big)$. Note that the rewriter must also add attributes to the graph.

Layer and Transformation Execution. Describing the complexity of DSLTrans transformation execution rests almost entirely on the matching and rewriting steps, as the complexity discussion for each layer and the transformation itself is simple.

For each layer, each rule must be executed over the input model to that layer. Therefore, the complexity of applying a layer in the transformation is the sum of all the matching and rewriting steps for each rule r in the layer over the input model g:

$$\mathcal{O}\big(applyLayer\big) = \sum |r_i|^{|g|}$$

For the execution of the transformation, the complexity is then the sum of the operations for each layer l in the transformation t:

$$\mathcal{O}\big(applyTransformation(t)\big) = \sum applyLayer(l) \quad \forall l \in t$$

3.4 Confluence and Termination Properties

This section will update the proof sketches for the confluence and termination of DSLTrans transformations from the thesis of Barroca [19] to ensure that they hold under the complete definition of DSLTrans in the double-pushout approach. Note that both these properties are enforced through the construction of the semantics of DSLTrans.

Note that the limited expressiveness of DSLTrans allows these proofs to be written in an intuitive and concise manner, rather than in a formal logic framework. More robust techniques concerning confluence and termination in more expressive languages can be found in Section 8.1.4 on page 278.

3.4.1 Confluence

This section will discuss the *confluence* property of the DSLTrans language. Confluence means that for every execution of a transformation $tr \in \textsc{Transforms}$ where the input-output model $iom_{pre} \in IOMs$ is the input model, the resulting input-output model iom_{post} is always the same up to typed graph isomorphism.

If two executions of the transformation were not confluent, then this would occur because of either conflicting rules, or during two potential points of non-determinism during the execution of the transformation.

Non-determinism is present in DSLTrans a) in the selection of rules in a layer, and b) in the selection of matches on the input graph, which could potentially give different results on different transformation executions. We shall explain here how the semantics of DSLTrans enforces confluence, despite the presence of non-determinism.

Proposition 1. *Confluence*

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

Proof. There are two areas to examine when discussing the property of confluence in DSLTrans. The first is the possibility of rules interfering with each other. The second area is the possibility of non-determinism in transformation execution.

1. Confluence for rule execution is determined by examining if all pairs of rules in the transformation which may conflict are in fact confluent [88]. These conflicting rules are termed *critical pairs*, and result from rules adding and deleting elements matched by the other [119].

   There are three situations where there may be conflicting rules. The first situation is where one rule deletes an element that the second rule could match over. This cannot occur in DSLTrans, as element deletion and attribute modification are not part of the language.

   The second situation for conflicting rules is where one rule would add elements to be matched by the second rule. However, as discussed in Section 3.2.5, both rules must match on the same input model to the layer, before they are applied in parallel. Consequently, rules cannot produce elements to be matched by another rule.

   The third situation is where elements produced by one rule interfere with the negative application graph (NAC) of a second rule. Again, as a rule cannot match over elements produced on the same layer, this situation cannot arise.

   Therefore, all rules are by construction *parallel independent* during execution of layer, including the repeated application of one rule. Thus no critical pairs of rules can exist in a DSLTrans transformation and rule application will be confluent.

2. The presence of non-determinism in the execution of a DSLTrans transformation does not affect the confluence of that transformation.

   The first point of non-determinism is in the application of a rule to an input-output model. Section 3.2.1 discusses how the double-pushout approach can be used to formalize the application of a rule by finding a match site. As mentioned in Ehrig *et al.*, the double-pushout approach itself is non-deterministic as multiple match sites may be found for each rule application [50]. However, in the DSLTrans approach (equivalent to a *layered graph grammar*), all match sites are found at once, *before* the rule elements are added in the rewriting step. This prevents selection of one matching site over another.

   However, this guarantee of application at all points does not hold in the presence of *Exists* elements in the DSLTrans rules (Section 3.2.6). Recall that the matching site for a particular *Exists* element is chosen from all suitable candidates in the target graph. If this matching site differs for two transformation executions, then this will violate confluency. It is therefore necessary to specify that a deterministic procedure is used to determine the matching site for *Exists* elements in all implementations of DSLTrans. This will ensure that executions of transformations is consistent and repeatable to preserve confluency.

   The second point of non-determinism in a transformation involves the selection of which rule to apply next in a transformation layer. In other model transformation languages, this selection process may cause violation of confluency.

   However in DSLTrans, the application of rules in a layer is modelled as a parallel graph production of all the rules at once, as discussed in Section 3.2.7. This means that rules cannot match over elements produced by other rules in the layer. Thus, all rules in a layer are parallel independent by construction and confluency is maintained, given repeatable matching for *Exists* elements. As well, the deterministic ordering of the layers within the transformation ensures that DSLTrans transformations will always have the same layers execute in turn.

Therefore, two executions of a DSLTrans transformation are indeed confluent up to typed graph isomorphism.

□

3.4.2 Termination

The *termination* property is crucially important for the analysis of transformations, such that the entire state space can be examined [87]. In particular, our research for verification of DSLTrans transformations discussed in Chapter 4 requires that the execution of transformations is finite.

Proposition 2. *Termination*

Every application of DSLTrans transformation must terminate.

Proof. Let us assume that there is an application of a DSLTrans transformation which does not terminate. For this to happen, there must exist some step in the transformation which may run indefinitely.

Here, we discuss four possible areas in the application of the transformation where infinite operations may be found:

1. The finiteness of a rule's matching step is given by the finiteness of the pattern and target graphs. Finding graph morphisms may be computationally expensive (see [43]), but it is a finite process as the number of vertices and edges are finite.

   Also recall that the transitive closure of graphs as required for the matching of indirect links is expressly prohibited from creating an infinite number of edges, as in Definition 6 on page 39, the transitive closure paths must contain unique nodes.

2. The matching and rewriting process of a rule production is finite in the double-pushout approach, as it only consists of finding the match and rewriting of the graph by adding elements.

   One particular area of concern is in the construction of a mass production, especially in the context of multiple rule application as in Section 3.2.5.1. Note that the rule matcher is finite, as is the target graph to be matched on. Thus even if a large derivation is created by considering multiple productions from the same rule, it is impossible for a match to be successful when the number of edges in the pattern graph is greater than the number of edges in the target graph. Therefore, all viable matchers must be of finite size.

3. Mass productions created from multiple rules must also be finite, as Definition 10 specifies that a number of rules in a layer is finite.

4. Definition 18 specifies that a transformation is composed of a sequential and finite list of layers. No looping constructs are permitted and thus the transformation moves through the layers sequentially.

Given that the semantics of rule and layer application are finite, and that transformation execution cannot loop through layers, every DSLTrans model transformation must terminate. $\square$

3.5 Conclusion

This section has presented the structures and semantics for the DSLTrans transformation language, grounded in the double-pushout approach. This approach allows for the matching and rewriting processes for rules in a layer to be elegantly expressed as a single production. This production operates on an input-output model to match and apply all elements in the rules at once, avoiding rule interference.

As a contribution of this thesis, the following constructs have been given semantics: *Any*/*Exists* elements, attributes on elements, negative application conditions (NACs), and indirect links. Thus, the formalization given in this section now covers all of the constructs of DSLTrans. This is required for our transformation verification research in Section 4.4 to cover all possible DSLTrans transformations.

As well, we have also presented a brief discussion of the complexity of executing a DSLTrans transformation to indicate the scalability concerns of DSLTrans. Finally, proofs for the confluence and termination properties of DSLTrans were presented, showing that DSLTrans transformations are guaranteed to terminate and provide repeatable output models in all cases.

Chapter 4
Transformation Verification using Contracts

This chapter discusses our symbolic-execution verification technique for DSLTrans transformation. In particular, we address two research questions from Section 1.1.1:

— RQ2: *How can the infinite execution possibilities of a transformation be represented in an explicit finite set?*

— RQ3:

  — a) *How can structural pre-condition/ post-condition contracts be proven to be satisfied or non-satisfied on these representations?*

  — b) *When a contract is not satisfied, how do the counter-examples produced relate to the transformation?*

Our approach to RQ2 relies on the concept of symbolic execution to construct *path conditions*, which represent all feasible executions (termed *input-output models*) of a DSLTrans transformation in a finite set. This representation is done through an abstraction relation, which we define in Section 4.1. This abstraction relation allows each path condition to represent an infinite set of possible input-output models. A number of abstraction relation examples are presented, including the abstraction of a rule applying multiple times and the use of indirect links in rules.

The use of the novel split morphism (Section 2.3.2.2) allows for path conditions to represent multiple input-output models without an explicit ‘disambiguation’ step. This disambiguation step was present in earlier research [98], where all possible combinations of elements within rules were produced. However, this is computationally infeasible as this operation explodes the state space.

We present arguments that our abstraction relation allows path conditions to both *represent* and *cover* the infinite set of input-output models in Section 4.3. Representation means that each path condition will abstract at least one input-output model, while coverage means that each possible input-output model is covered by at least one path condition.

Section 4.2 then discusses the path condition generation process, which resolves the dependencies of rules to create only those path conditions which represent valid input-output models. As a contribution of this thesis, an intuitive explanation of path condition construction is provided which includes negative elements in rules and removal of invalid rule combinations, allowing the technique to apply to all DSLTrans transformations.

As well, this thesis also examines the critical issue of rule multiplicity in path conditions. As it is impossible to represent an infinite application of rules within the path condition, it is necessary to abstract over the number of times each rule applies. This chapter examines how to explicitly represent each application of rules within the path condition, and the consequences for path condition construction and contract proving.

After path condition generation, structural contracts can then be proved on these path conditions to determine those where the contract is and is not satisfied. Thus, RQ3 is answered when we discuss contract verification in Section 4.4. Through the abstraction relation, if a contract is satisfied or not on a path condition, this has a relationship to how the contract is satisfied or not on the abstracted input-output models. This allows the transformation verifier to understand the relationship between input and output models for the transformation by examining the results of contract proving. The validity of the contract proving with respect to the abstraction relation is discussed in Section 4.5.

Finally, we discuss the expressibility of the contract language in Section 4.6. This section provides an overview of the constructs in the contract language which allows for constructing propositional statements with contracts. Examples from our case studies (Section 7) are employed to demonstrate the semantics and limitations of the contract language.

4.1 Abstraction Relation

For a DSLTrans transformation, there are an infinite number of input-output models possible, each one composed of a pair of an input model to the transformation and the resulting output model. This section details our abstraction relation which allows us to create 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.

In particular, this section defines the abstraction relation between a path condition and the set of input-output models that it represents. This abstraction relation allows our technique to prove contracts on the finite set of representative 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.

4.1.1 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*" [81], 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.

Our technique transposes this notion of symbolic execution to model transformations. The analogue of an input variable in the model transformation context are *meta-model classes*, *associations* and *attributes*. Just as program statements impose constraints on input and output variables during symbolic execution, transformation rules impose conditions on which meta-model elements are instantiated during a concrete transformation execution, and how that instantiation happens. As well, 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 transformation executions as there are input-output models that satisfy them. This is formulated as an *abstraction relation* as defined in Section 4.1.4 on page 87.

4.1.2 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 step or *production* as discussed in Section 3.2.1), by recording the input and output elements induced by rule application.

As described in Section 2.2, a layer in a DSLTrans transformation contains a set of rules. Let us begin by creating a set of rule combinations for this layer, where each rule combination in this set will represent all possible input-output models where the rules in that combination would execute.

For example, in Figure 4.1, the rule combination marked AB represents the set of input-output models with at least one application of rules A and B. Another rule combination marked A represents the input-output models where only rule A has applied at least once. Another rule combination is marked AA representing at least two applications of the A rule.

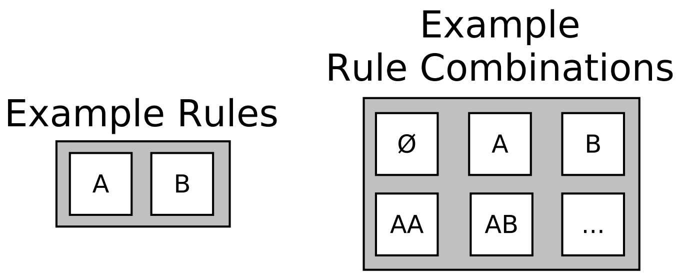


Figure 4.1 – Rule combinations created for a transformation layer.

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 input-output models. These path conditions will explicitly contain the input and output elements induced by rule application, as demonstrated in Section 4.1.3.

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 2.2, 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, as this would create invalid combinations. Instead, our approach must move layer-by-layer and resolve the dependencies between rules.

4.1.3 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.

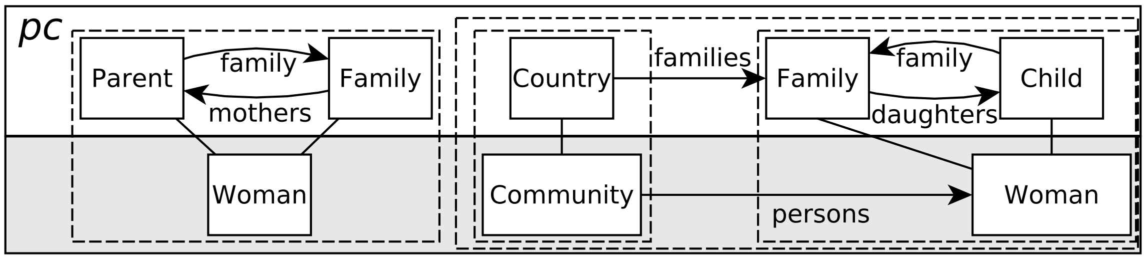


Figure 4.2 – An example path condition representing the symbolic execution of four rules.

For example, the graphical representation of a path condition in Figure 4.2 represents the application of four rules in the *Families-to-Persons* transformation from Section 2.2: (*Mother2Woman*, *Country2Community*, *Daughter2Woman*, and *copersons...Woman*). This representation includes the input and output elements that will be present in the input and output models if these four rules apply in the transformation. The input elements are those in the top half of the path condition, while the bottom half contains the output elements. Traceability links connect input and output elements within rules, which are represented by overlapping dashed boxes.

As an example, in Figure 4.2 the *Mother2Woman* rule on the left consists of two input elements *Parent* and *Family*, connected by associations. These elements are connected to the *Woman* output element by traceability links. Therefore, this path condition indicates that when there are *Parent* and *Family* elements in the input model to the *Families-to-Persons* transformation, then a *Woman* element will be produced in the output model.

Note that the path condition in Figure 4.2 is representing a lower bound on the number of times those rules have applied on an input model. This means that this path condition also represents those input-output models where the four rules have applied any number of times, thereby matching and producing further input and output elements. An example of an input-output model that this path condition abstracts over is seen in Figure 4.11 on page 102.

Definition 19. *Path Condition*
A path condition is a three-tuple $\big\langle Input,\ Output,\ trace,\ V_{Neg},\ E_{Neg},\ \big\rangle$, *where:*

— $Input, Output \in \mathrm{TG}$

— *Input and Output may be the empty typed graph* $\emptyset$

— $Input \cap Output = \emptyset$

— $trace = \langle E_{trace}, (s_{trace}, t_{trace})\rangle$*: the traceability links*

  — E_{trace} *is disjoint from* E_{Input}*, and* E_{Output}

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

— V_{Neg} *is the set of negative vertices in the Input graph*

— E_{Neg} *is the set of negative edges in the Input graph*

The empty path condition is defined as $\epsilon_{pc} = \{\emptyset, \emptyset, \langle\emptyset, (s_{trace}, t_{trace})\rangle\}$.
The set of all path conditions for a transformation tr is $\textsc{PathConds}_{tr}$.

A formal definition of path conditions is presented in Definition 19. Note that the structure of path conditions is intentionally very similar to that of DSLTrans rules and input-output models (Section 3.1).

The *Input* graph of a path condition represents those elements that are present (or not present) in the input model of the transformation for the path condition to hold. The *Output* graph of a path condition contains those elements which will be present in the output model of the transformation if the *Input* graph is present in the input model. In other words, if all the elements in the *Input* graph are found in an input model of the transformation, then all the elements in the *Output* model will be present in the respective output model of the transformation.

Traceability links are present in E_{trace} between elements in the *Output* and *Input* graphs to replicate the traceability information created during application of DSLTrans rules, as discussed in Section 2.2.

Note that as the path condition represents the execution of DSLTrans rules, a path condition may also contain negative elements in the *Input* graph. This is to indicate that these elements are not present in the input model of the transformation. Figure 4.10 on page 101 shows an example of how these negative elements can prevent a path condition from abstracting over an input-output model.

We define a utility function GETRULES: $V_{Input} \rightarrow \mathcal{P}\{Rules\}$. This function accepts a vertex from the *Input* graph of the path condition and returns the set of rules which that vertex represents. For example, let f be the *Family* element on the right side of the path condition in Figure 4.2. Then, GETRULES(f) would return $\{Daughter2Woman, copersons\ldots Woman\}$.

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, which is needed for reporting the status of contract verification (Section 4.4).

4.1.4 Definition of the Abstraction Relation

The *abstraction relation* defined in Definition 20 is at the core of our symbolic execution technique. It allows us to represent an infinite set of input-output models with a finite set of path conditions. Contract proof can then take place on this set of path conditions and the results will be extended to the set of input-output models, as further explored in Section 4.4.

As can be seen in Figure 4.2 on page 85, path conditions represent a set of input-output models by storing the elements that will be present in the input and output models for that execution, along with traceability links which track how the elements were created. The abstraction relation provides the mapping so that these path condition elements will be present in the transformation execution, and vice versa.

In other words, the path condition in Figure 4.2 contains the input and output elements present when those four rules from the transformation apply. Our abstraction relation states that for all possible input-output models where only those four rules apply (any number of times) those input elements will be present in the input model, and those output elements will be present in the output model.

Specifically, the abstraction relation will map:

— all input elements and traceability links in the path condition to the input-output model;

— all output elements and traceability links in the input-output model to the path condition.

Therefore, if the abstraction relation holds for a particular path condition and input-output model, there is a guarantee on the elements present in the input-output model. This information will then be used during contract proof as described in Section 4.4 on page 126.

4.1.4.1 Formal Definition

This section will formally define the notion of abstraction of an input-output model by a path condition. This is done through the discussion of five conditions which are required for correctness and consistency of the abstraction relation. After the conditions are presented, then Figures 4.4 to 4.11 on pages 94–102 will present examples of the abstraction relation holding and not holding.

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 20. *Abstraction of a Input-Output Model by a Path Condition*
Let tr be a transformation. Let iom = $\langle Input_{iom},\ Output_{iom},\ trace_{iom} \rangle \in \mathrm{IOM}_{tr}$ *be an input-output model for that transformation.*

Let also pc = $\langle Input_{pc},\ Output_{pc},\ trace_{pc},\ V_{Neg},\ E_{Neg},\ \rangle \in \textsc{PathConds}_{tr}$ *be a path condition for that transformation.*

Abstracts: *We have that pc abstracts iom, denoted* $pc \mapsto iom$*, if and only if the following conditions are all true:*

Condition 1: pc_{Input} semi-injectively matches onto iom_{Input}.
The purpose of the first condition, defined in Equation 4.1, is to enforce that the path condition accurately represents the elements present in the input model of the input-output model. Therefore, the elements in the path condition's *Input* graph are searched for in the *Input* graph of the input-output model.

$$\begin{aligned} &\exists f : \big(Input_{pc} \underset{f}{\rightarrow} Input_{iom}\big) \text{ where} \\ &\forall v_1, v_2 \in V_{Input_{pc}} : \big(f(v_1) = f(v_2)\big) \implies \big(getRules(v_1) \cap getRules(v_2) = \emptyset\big) \end{aligned} \quad (4.1)$$

Note that the morphism f we are attempting to find in Equation 4.1 is not necessarily injective. This is because an important consideration for the abstraction relation is that the path condition is constructed through a composition of rule elements. Therefore, each *rule component* in the path condition must be found in the input-output model, but elements from different rules may match onto the same element in the transformation execution.

This restriction on elements matching from the same rule is presented in the last line of Equation 4.1, where elements that overlap in the input-output model cannot belong to the same rule in the path condition, as given by *getRules*. This condition can be seen in the fourth example in Figure 4.6 on page 98, where rule components overlap in a successful abstraction.

Note that this definition allows for extra elements that may be present in the input-output model but not matched over by the path condition. These extra elements represent elements not matched by the rules which symbolically applied in the path condition. This case is seen in Figure 4.4, where the empty path condition abstracts over an input-output model with extra input elements.

As well, it is important to note that the *Input* graph of the path condition may contain negative elements. In this case, the morphism f between the path condition and the input-output model is extended to match both the PAC and NAC, which is extracted from the *Input* graph as in Section 3.2.2. An example of negative elements in the path condition is presented in Figure 4.10 on page 101.

Condition 2: iom_{Output} **surjectively matches onto** pc_{Output}**.**

The surjection relation g defined in Equation 4.2 enforces that all elements in the *Output* graph of the input-output model must be matched onto the *Output* graph of the path condition. The surjective nature of the relation means that each element in the path condition *Output* graph must be matched by at least one element.

$$\exists g : Output_{pc} \overset{surj}{\underset{g}{\blacktriangleleft}} Output_{iom} \tag{4.2}$$

This condition represents the constraint that elements produced by execution of the transformation (the *Output* graph of the input-output model) must be represented by a path condition that contains those output elements.

Note that there may be multiple copies of an element in the *Output* graph of the input-output model, but the path condition represents the production of the element 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 allows for a path condition to represent rules applying multiple times, without explicitly representing each application.

Examples of this rule multiplicity abstraction are presented in Figure 4.7 on page 99.

Condition 3: pc_{trace} **semi-injectively matches onto** iom_{trace}.

The purpose of the third condition is to ensure that the rule application represented in the path condition reflects the actual rule application performed in the input-output model. As defined in Equation 4.3, traceability links in the path condition must injectively match onto links in the input-output model using the morphism j.

$$\exists j : trace_{pc} \overset{link}{\underset{j}{\rightarrow}} trace_{iom} \mid \forall e_1, e_2 \in trace_{pc} : \\ j(e_1) = j(e_2) \implies getRules(e_1) \cap getRules(e_2) = \emptyset \quad (4.3)$$

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

Condition 4: iom_{trace} **surjectively matches onto** pc_{trace}.

Traceability links in the input-output model must surjectively match onto links in the path condition with the morphism k defined in Equation 4.4.

$$\exists k : trace_{pc} \overset{link}{\underset{k}{\blacktriangleleft}} trace_{iom} \quad (4.4)$$

This condition is to ensure that there is no rule application in the input-output model which is not represented by the path condition. If there is a link that is not found in the path condition, then this path condition does not represent the application of that rule.

Condition 5: Consistency.

The morphisms (f,g,j,k) used in the above conditions must be consistent with each other and agree over common elements.

For example, if a traceability link in the path condition is matched to a traceability link in the input-output model with one morphism, then the source and target of those traceability links must be matched together with the other morphisms.

We refer to Section 2.3.2.5 on page 40 for the precise definition of consistency, Equation 4.5 and Equation 4.6 extend this definition to all four morphisms.

$$trace_{pc} \overset{link}{\underset{j}{\rightarrow}} trace_{iom} \implies \forall l \in trace_{pc} : \big(f(t(l)) = t'(f(l))\big) \tag{4.5}$$

$$trace_{pc} \overset{link}{\underset{k}{\blacktriangleleft}} trace_{iom} \implies \forall l \in trace_{iom} : \big(g(s(l)) = s'(g(l))\big) \tag{4.6}$$

As an example of consistency in the abstraction relation, consider the example of rule multiplicity in Figures 4.7 on page 99. Let the elements $tlink_{pc}$, A_{pc}, and X_{pc} denote the traceability link and connected A and X elements in the path condition. As well, let their counterparts in the input-output model be denoted $tlink_{iom}$, A_{iom}, and X_{iom}.

Note that two morphisms from the path condition involve these elements, the element morphism f and the traceability link morphism t. The consistency condition therefore requires that if the traceability link morphism j maps $tlink_{pc}$ to $tlink_{iom}$, then the element morphism f must map the A and X elements to their counterparts. That is, $j(tlink_{pc}) = tlink_{iom} \implies \big(f(A_{pc}) = A_{iom} \wedge f(X_{pc}) = X_{iom}\big)$.

4.1.5 Examples

In this section, we provide a number of examples to demonstrate different facets of the abstraction relation:

1. Empty Path Condition
   — The empty path condition abstracts all input-output models where no rules apply.
2. Non-overlapping Rule Components
   — Different rules do not contain similar elements so the matching is injective.

3. Overlapping Rule Components
   — Different rules do contain similar elements so the element matching is semi-injective.
4. Multiple Rule Application
   — Rules apply multiple times in the path condition and/or the input-output model.
5. Indirect Links
   — Indirect links are present in the path condition.
6. Negative Elements
   — Negative elements are present in the path condition.
7. *Families-to-Persons* Example
   — A path condition and an input-output model from the *Families-to-Persons* transformation are presented.

In each figure in this section, the path condition *pc* is on the left-hand side, and an input-output model *iom* is on the right-hand side.

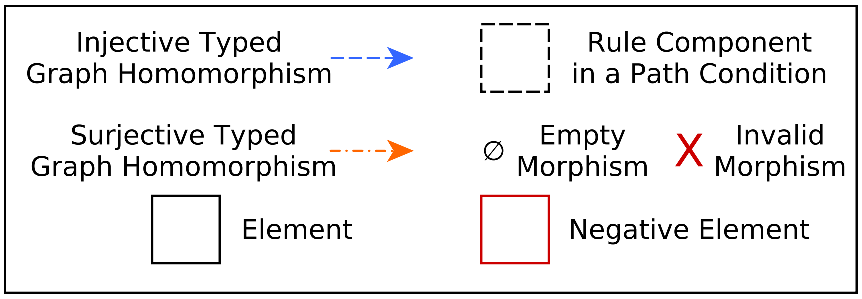

Injective Typed Graph Homomorphism
Rule Component in a Path Condition
Surjective Typed Graph Homomorphism
∅ Empty Morphism
X Invalid Morphism
Element
Negative Element

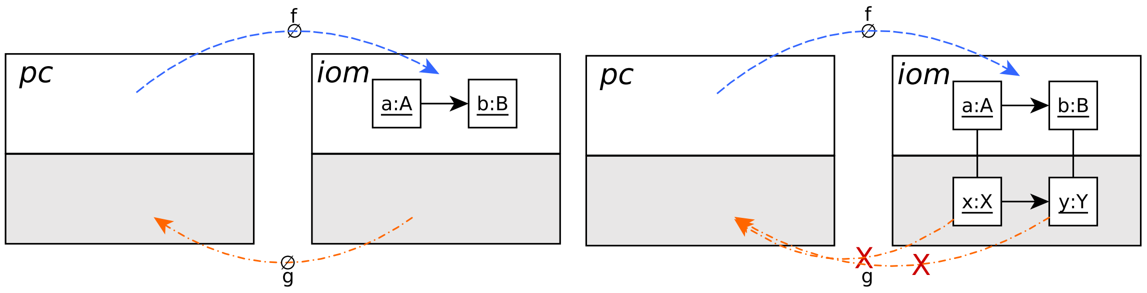
Figure 4.3 – Legend for abstraction relation figures.

Figure 4.3 presents the legend for the morphisms in the following figures. These morphisms will show potential matches between elements. An X placed over the morphism arrow means it cannot be satisfied, while an $\emptyset$ symbol means that an empty morphism was found. The presence of these matches allow the conditions of the abstraction relation to hold.

We also note that the abstraction relation must also match over the associations in the path condition and input-output model. However, this matching is not included in the figures to improve visual clarity.

4.1.5.1 Example 1 – Empty Path Condition

The first example in Figure 4.4 demonstrates how an empty path condition will abstract over all input-output models where no rules apply. In Figure 4.4a, the empty path condition abstracts the input-output model. However, in Figure 4.4b the empty path condition does not abstract the input-output model because of rule application in the input-output model producing elements not found in the path condition.



(a) Abstraction relation holds. (b) Abstraction relation does not hold.

Figure 4.4 – Abstraction relation of input-output models by the empty path condition.

The *Input* graph 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 applied.

The first condition for the abstraction relation is to determine whether a morphism can be found between the *Input* graph of the path condition and the *Input* graph of the input-output model. Note that in both Figure 4.4a and Figure 4.4b, an empty morphism satisfies this condition, denoted by the blue arrows and ∅ symbol.

The second condition for the abstraction relation is whether a surjective morphism can be found from the output model to the output graph of the path condition. This is represented by orange arrows in Figure 4.4a and Figure 4.4b. 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* graph of the input-output model, as no rules have executed. Note that an (empty) surjective morphism can be found between the *Output* graph of the input-output model in Figure 4.4a and the path condition. This is intuitive as the lack of elements in the *Output* graph of the input-output model means no rules have executed, which corresponds to the lack of a post-condition defined by the path condition.

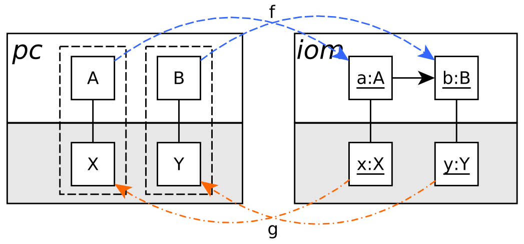
In contrast, in Figure 4.4b no surjection morphism can be found between the elements of the *Output* graph of the input-output model and the path condition. As the input-output model has elements in the *Output* graph, at least one rule must have applied. However, the path condition does not represent that a rule has applied and thus it 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 models where the input model does not contain elements sufficient for any transformation rules to apply. As well, all input-output models not abstracted by another path condition will necessarily be represented by the empty path condition. This is employed by our proof sketches in Section 4.3.2 to argue that all input-output models are abstracted by a path condition.

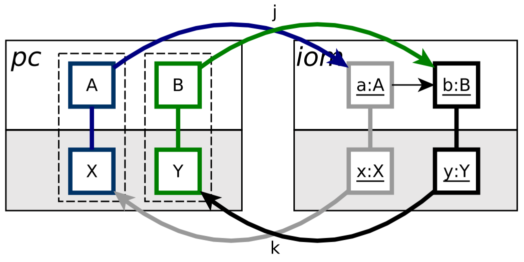
4.1.5.2 Example 2 – Non-overlapping Rule Components

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 conditions of the abstraction relation, two figures are displayed for the following examples. The first will demonstrate the matching performed on elements for both the input and output graphs, while the second figure focuses on the matching of the traceability links.



(a) Abstraction relation holds on non-overlapping elements.



(b) Abstraction relation holds on traceability links.

Figure 4.5 – Abstraction relation for non-overlapping rule components.

Let us first examine how the morphism operates between the *Input* elements in the path condition and the input-output model in Figure 4.5a. Note that the morphism can be found from the elements in the path condition's *Input* graph successfully. Similarly, there is a surjection morphism that can be found between the *Output* elements of the input-output model and the output graph of the path condition.

We now examine Figure 4.5b to resolve whether the traceability links (and connected elements) in the path condition can be found in the input-output model. This matching is represented by the arrows, where each link is highlighted in a bold outline and differentiated by colour. We note that each traceability link 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. In Figure 4.5b the links in the input-output model are successfully matched onto the path condition.

Finally, we note that consistency is maintained in Figure 4.5. The source and target of each traceability link are matched to the connected elements in a consistent manner with the other morphisms.

4.1.5.3 Example 3 – Overlapping Rule Components

For this example, we allow path conditions which 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 4.6a correctly match over the input-output model shown. The abstraction relation holds due to the fact that, while the elements of the same component cannot match over the same element in the input-output model, elements from different rules do not have any constraint.

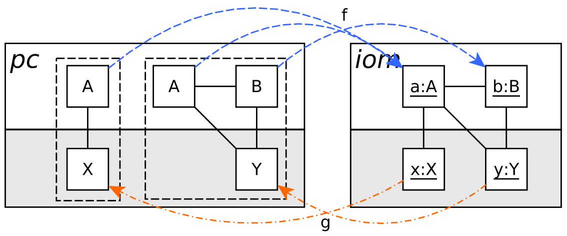
In other words, the path condition in Figure 4.6a represents the application of two rules which may or may not match over the same element in the *Input* graph of the input-output model.

The flexibility of this matching allows our technique to avoid having to create 'disambiguated' path conditions where this combination of elements is explicitly produced, as in the thesis of Barroca [19].

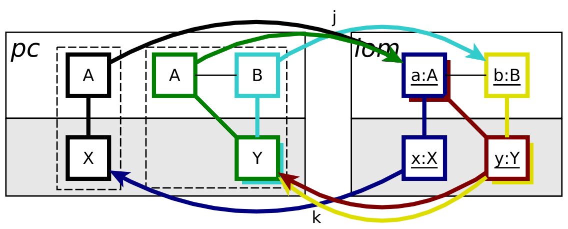
Figure 4.6b shows the mapping from the traceability links of the path condition to the input-output model. Each traceability link and the connected components are matched and found. The traceability links from the input-output model are also matched back onto the path condition.

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 as denoted by the three arrows in the bottom of Figure 4.6b.

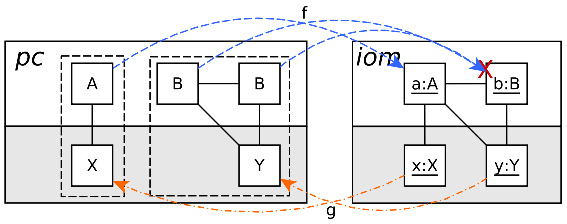
In contrast, Figure 4.6c shows an example where the abstraction relation does not hold. Note that one rule component in the input graph of the path condition contains two B elements. Both of these elements must be matched for the rule to



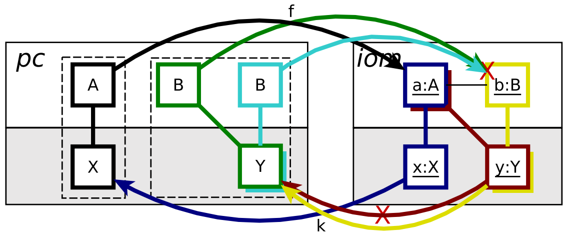
(a) Abstraction relation holds with overlapping elements.



(b) Abstraction relation holds on traceability links.



(c) Elements cannot overlap within a component.



(d) Traceability links cannot be found.

Figure 4.6 – Abstraction relation with overlapping rule components.

apply, and thus it is not correct for them to match to the same element in the input-output model. Thus, a mapping cannot be found and the path condition does not abstract the input-output model.

As well, it is informative to examine Figure 4.6d. 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 in the path condition. Therefore, this path condition cannot accurately represent this input-output model.

4.1.5.4 Example 4 – Multiple Rule Application

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.

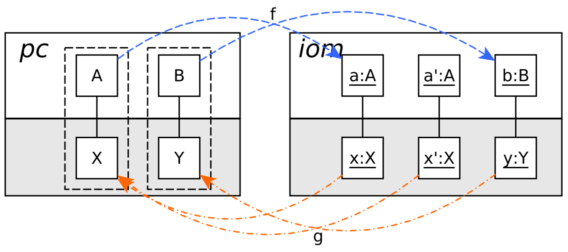
Recall from Section 4.1.3 that rule application is represented as either not occurring, or as a lower bound in the path condition. That is, if a rule does not appear in a path condition, the rule does not apply in the input-output model. If the rule appears once, it applies one or more times in the input-output model. If the rule appears twice, it applies two or more times, and so on.

Figure 4.7 shows an example of the abstraction relation where the rule applies multiple times in the input-output model.

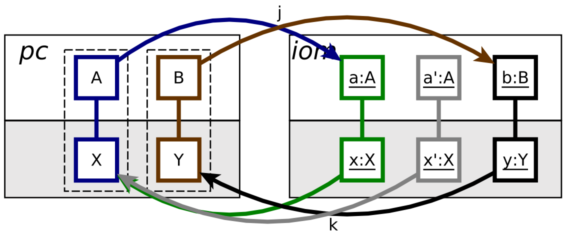
Note that the elements of multiple copies of the rule in the path condition can be matched over the input-output model in Figure 4.7a, as long as the rule multiplicity in the path condition is a lower bound for the input-output model.

The surjective match also holds in Figure 4.7b, where one rule has applied twice in the input-output model but only once in the path condition. The presence of extra rule application in the input-output model is handled by the surjective nature of this match, as there does not need to be a unique (injective) mapping.

Note that the consistency condition is required to ensure that the f and j morphisms select the same elements (the a' $\rightarrow$ x' elements).



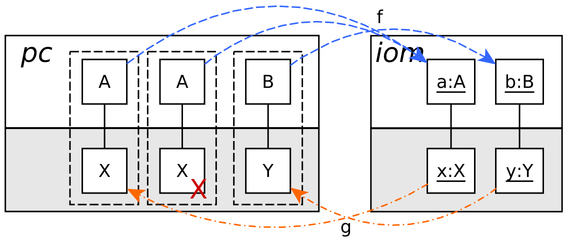
(a) Representing a lower bound of rule application.



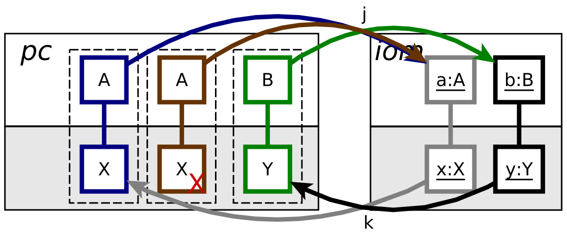
(b) Traceability links surjectively match on the input-output model.

Figure 4.7 – Example of successful abstraction relation with multiple rule application.

In contrast, Figure 4.8 demonstrates how the abstraction relation will not hold if the rule is applied fewer times in the path condition than the input-output model. This is because the elements from the duplicated $a \rightarrow x$ rule cannot be surjectively matched over by the output elements in the input-output model.



(a) Cannot surjectively cover more applications than are in the input-output model.



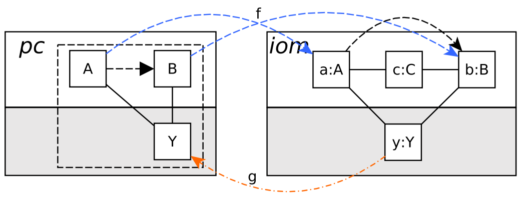
(b) Traceability links also cannot be covered.

Figure 4.8 – Example of unsuccessful abstraction relation with multiple rule application.

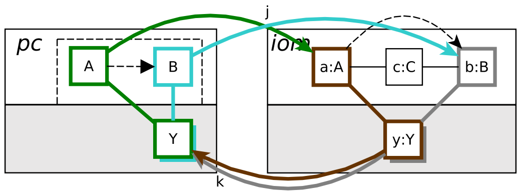
The traceability link matching shown in Figure 4.8b also demonstrates how it is not possible to find an surjective match in this case. Thus, a path condition which represents more rule applications than the input-output model cannot abstract over that input-output model.

4.1.5.5 Example 5 – Indirect Links

Figure 4.9 presents a path condition that includes rules with indirect links, where the links are shown as dashed lines between the elements. 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 Section 2.3.2.3. This closure creates links between all nodes connected by a non-looping path.



(a) Matching on the link created.



(b) Matching over traceability links.

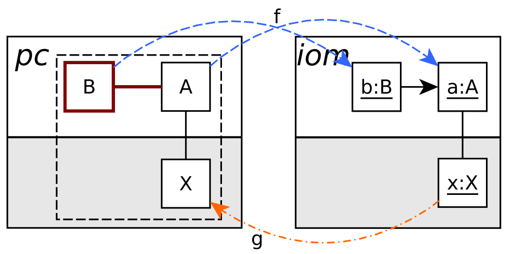
Figure 4.9 – Abstraction relation with indirect links.

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.

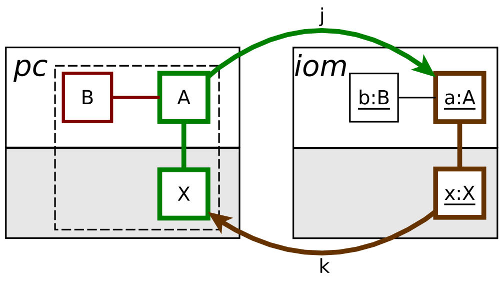
4.1.5.6 Example 6 – Negative Elements

We show an example of a path condition in Figure 4.10 where the path condition represents rules containing negative elements.

When negative elements are present in a rule component, we rely on the same mechanism as for rule application that was discussed in Section 3.2.2. The path condition's *Input* graph is divided into a positive application graph (PAC) and a negative application graph (NAC) using the extraction process presented in Section 3.2.2. Then, for the path condition to abstract the input-output model, the PAC must match over the input model, and the NAC must not match.



(a) Cannot match the negative elements.



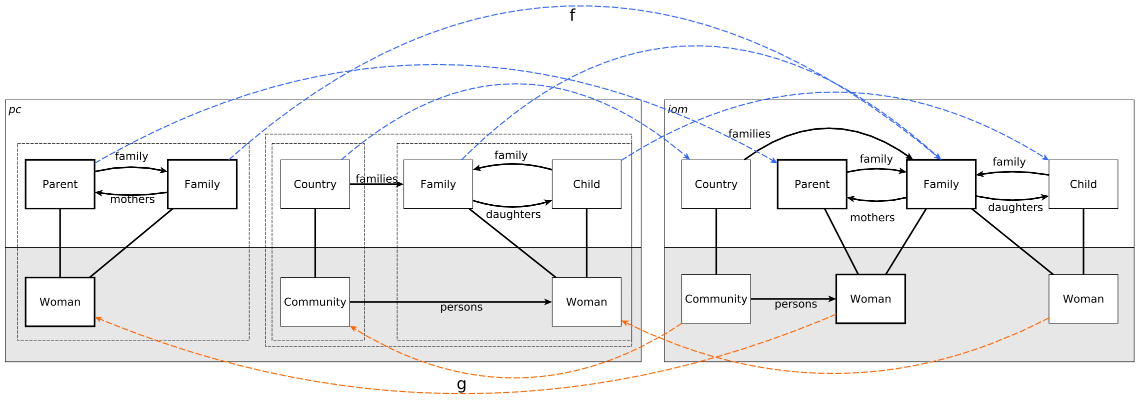
(b) Traceability links do match.

Figure 4.10 – The negative element makes the abstraction relation not hold.

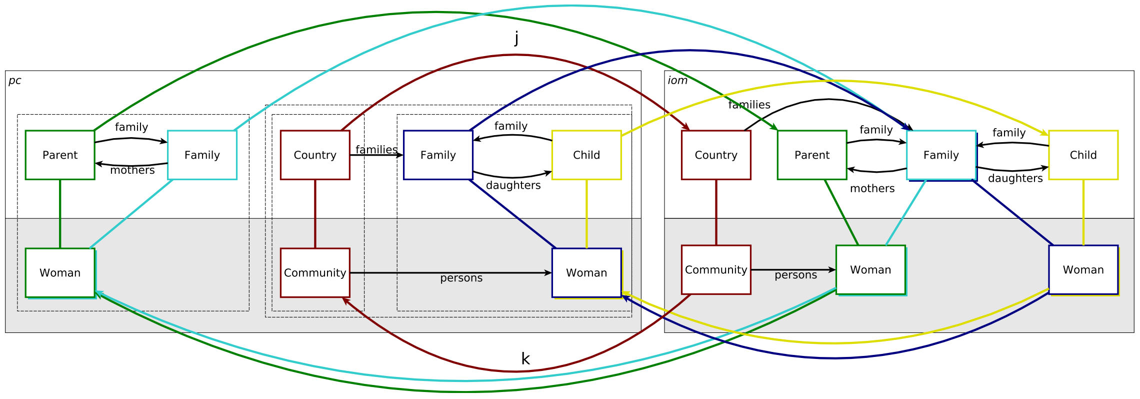
In Figure 4.10, the PAC consists of the A element, while the NAC consists of the B element, the A element, and the connecting association. Note that in this case, both the PAC and the NAC match, so this path condition cannot abstract this input-output model.

4.1.5.7 Example 7 – *Families-to-Persons* Example

Finally, Figure 4.11 demonstrates the abstraction of an input-output model in the *Families-to-Persons* transformation (Section 2.2) by a path condition. Note that



(a) The elements of the path condition are found in the input-output model



(b) Traceability links match as well.

Figure 4.11 – Abstraction of a path condition on an input-output model for the *Families-to-Persons* transformation.

the two *Family* elements in the path condition are matched onto the same *Family* element in the input-output model.

This particular abstraction example will become important when we discuss how a contract is satisfied by both a path condition and an input-output model in Section 4.4.2.

4.1.6 Other DSLTrans Constructs

We must also consider the presence of other DSLTrans constructs to be able to handle all DSLTrans transformation. Namely, this section discusses attributes and *Any*/*Exists* elements as introduced in Section 2.2.5.

Attributes. The rules in DSLTrans may also include attributes on the match or apply elements in rules. As path conditions are composed of rule components, these attributes will also be present on path condition elements.

Therefore, we take a similar approach to attribute resolution as in Section 3.2.3 on page 59. We assume that there is a proper solver that can determine the status of this attribute matching. Then, for elements in the path condition to abstract over the elements in the input-output model, we submit these elements to the resolver to determine if the matching succeeds.

Any/Exists Elements. The *Any*/*Exists* elements denotes different semantics for the matching of multiple executions of the same rule, as discussed in Section 3.2.6 on page 65.

These semantics are represented in the abstraction relation by restricting the matching of *Exists* elements in the path condition over elements in the input-output model. This restriction is on the morphism f in the first condition of Definition 20. Specifically, *Exists* elements in multiple applications of a rule in a path condition must be resolved as in Section 3.2.6.

4.1.7 Partitioning of Input-Output Models

The abstraction relation partitions the infinite set of input-output models 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 input-output models.

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 pc could abstract all input-output models where rule A applies at least once, while another path condition pc' could abstract all input-output models where rule A applies two or more times. Thus, the set of input-output models abstracted over by pc' would be a subset of those abstracted by pc.

This overlapping has two consequences. First, there will be an ordering of path conditions based on the abstraction of the multiplicity for rule application. Second, an input-output model may be abstracted by multiple path conditions. In Section 4.5.4 on page 142 we will discuss how this ordering affects the contract verification process, which relies on the abstraction relation to extend the contract proof result on a path condition to the input-output models it abstracts.

4.2 Building Path Conditions

This section describes our symbolic execution approach for building path conditions. The algorithm examines interactions of rules, such that all rules in the transformation are combined into path conditions by examining each layer in the transformation. At the end of this process, each path condition will abstract over an infinite set of input-output models using the abstraction relation presented in the previous section.

Note that the current implementation of this approach in our contract proving tool SyVOLT is discussed in Section 5.3.4 on page 188.

4.2.1 Path Condition Generation Overview

The aim of the path condition generation process 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 4.12.

This process will begin with *PCSet* as containing only the empty path condition. Then, the path conditions in *PCSet* will be iteratively combined with rules from the next layer to determine the symbolic execution of each rule. This combination

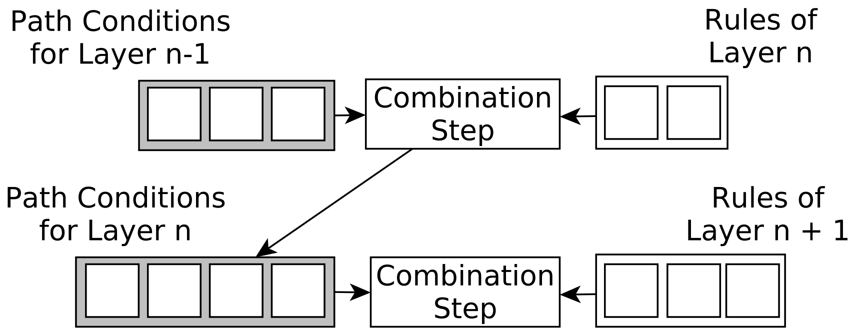


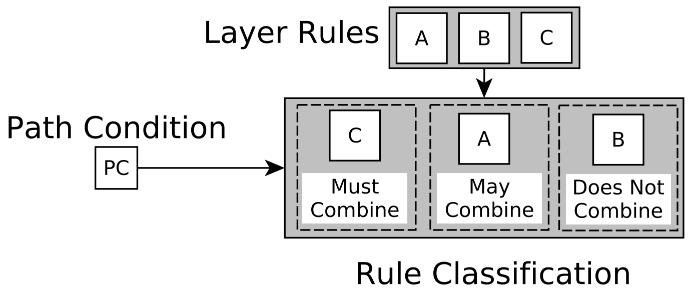
Figure 4.12 – Previous path conditions are combined with rules.

process will produce a new set of path conditions as the *PCSet* for the next layer. Once all layers of the transformation have been examined, a complete set of path conditions for the entire transformation will have been produced.

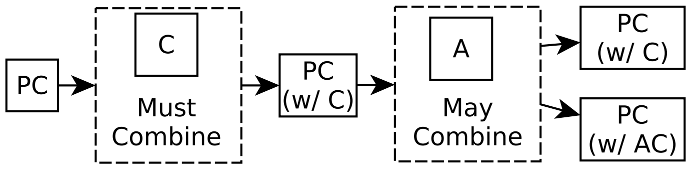
The *combination step* in Figure 4.12 is the core of our technique, where each path condition in *PCSet* is examined to determine how it combines with the current layer's rules. This combination is based on determining how the rules match over the input and output elements present in the path condition, and whether the rule *may* or *must* apply on that path condition. Based on this classification, which is explained in Section 4.2.2, new path conditions will be created which contain the rule's elements.

An example is shown in Figure 4.13a, where the rules in a layer are classified as *may combine*, *must combine*, or *does not combine* with a path condition. Figure 4.13b shows how new path conditions are created by the combination of the rules with that path condition.

In Figure 4.13, the *B* rule is discarded as it does not combine with this path condition. The *C* rule must combine with the path condition, while the *A* rule may or not combine. Thus, a path condition is created where the rule *A* is combined, and one where it does not. Note that this *may combine* 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 4.13 – The path condition creation process for one path condition.

4.2.2 Rule Classification Sets

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

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

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 **must** apply on pc

We add rl to the set $must_{pc}$ if rl *must* symbolically execute over pc, or rl will be added to the set may_{pc} if the rule *may* or *may not* execute over pc.

4.2.2.1 Rule Combination Matcher and Rewriter

The decisions for how a rule could combine with a path condition are based on whether there is a morphism that can be found between the rule's matcher and the path condition.

We construct this matcher – denoted $\lceil rl \rceil$ – for the rule rl by following Definition 13 on page 49. As explained in that section, this matcher is composed of the *Match* elements in the rule, along with backward links and any *Apply* elements connected to those backward links.

As mentioned in Section 2.2.4, 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 represented by the path condition.

Note that the rule matcher will also contain negative elements, which define dependencies on which elements are not present within the path condition.

The rewriter for the rule – denoted $\lfloor rl \rfloor$ – for the rule rl is constructed by following Definition 14 on page 50. However, we also retain in the path condition rewriter the negative elements in the rule. As described in Section 4.2.3.3, this allows the path condition to store negative elements.

4.2.2.2 Must Combine Set

Definition 21 defines the $must_{pc}$ set, which holds all rules which *must combine* with the path condition pc.

Definition 21. *Path Condition and Rule Combination – Must Combine*
Given a path condition pc and a DSLTrans layer l:

$$must_{pc} = \big\{ rl \in l \mid (rl_{back} \neq \emptyset \wedge \neg containsNegativeElements(rl) \wedge \lceil rl \rceil \rightarrowtail pc) \big\}$$

This definition selects those rules from the layer where a) the rules specifies dependencies through backward links, b) the rule doesn't contain negative elements (or associations), and c) the rule's matcher (which includes the backward links) matches over the path condition using the typed graph split morphism $\rightarrowtail$. Therefore, all elements required by the rule to execute are present in the path condition and thus in all abstracted transformation executions through the abstraction relation.

Note that if negative elements 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 may_{pc}.

4.2.2.3 May Combine Set

Definition 22 creates the set may_{pc} which holds all rules which *may combine* with the path condition *pc*.

Definition 22. *Path Condition and Rule Combination – May Combine*
Given a path condition pc and a DSLTrans layer l:

$$may_{pc} = \left\{rl \in l \mid rl_{back} = \emptyset \vee \left(rl_{back} \neq \emptyset \wedge rl_{back} \overset{link}{\rightarrow} pc\right)\right\}$$

This definition selects those rules where either a) the rule specifies no dependencies through its backward links, or b) there are dependencies specified and they match over the path condition.

Therefore, the set may_{pc} will contain rules where the dependencies specified are satisfied by the path condition, but all input elements *may* or *may not* be in the path condition. Therefore it may be possible that the rule applies for this path condition, but it may not.

4.2.2.4 Does Not Combine Set

The last rule set created, not_{pc} contains the rules that do not combine with the path condition. As shown in Definition 23, this set contains the rules not contained in $must_{pc}$ or may_{pc}.

Definition 23. *Path Condition and Rule Combination – Does Not Combine*
Given a path condition pc and a DSLTrans layer l:

$$not_{pc} = \{rl \in l \mid rl \notin must_{pc} \wedge rl \notin may_{pc}\}$$

Note that this set is created only for completeness, and the rules contained will not be combined with the path condition.

4.2.2.5 Creation of Path Condition Set

The steps in Section 4.2.3 will take a path condition *pc* and a set of rules, and produce a *must combine* mass production p^*_{must}, and a set of *may combine* mass productions p^*_{may}. These mass productions will combine the elements from the rules with the path conditions.

Algorithm 1 – Building path conditions for a layer.

Input: The set of path conditions from the previous layer, and the set of rules for the current layer
Output: The set of path conditions for this layer

```
 1: function CreatePathConditions(PCSet, rules)
 2:     newPCSet ← ∅
 3:     for each pc ∈ PCSet do
 4:         mustRuleSet ← findMustMatch(pc, rules)                ▷ $must_{pc}$
 5:         mayRuleSet ← findMayMatch(pc, rules)                  ▷ $may_{pc}$
 6:
 7:         mustMassProd ← createMassProd(pc, mustRuleSet)
 8:
 9:         oldPC ← copy(pc)
10:         pc ← apply(mustMassProd, pc)
11:
12:         mayPowerSet ← createPowerSet(mayRuleSet)
13:
14:         for each combo ∈ mayPowerSet do
15:             prod ← createMassProd(oldPC, combo)
16:             pcCopy ← apply(prod, pc)
17:             Add pcCopy to newPCSet
18:     return newPCSet
```

To create the new set of path conditions *newPCSet* to be presented to the next layer, these productions created are applied to the path condition. This application is performed in Algorithm 1, which takes the path conditions from the previous layer, and combines them with rules from the current layer.

Algorithm 1 begins by iterating through each path condition in the set. The first step is to divide up the rules in the layer into the *mayRuleSet* and the *mustRuleSet* on Lines 4 and 5. Then the *must combine* mass production can be created out of the rules in the total rule set on Line 7. This mass production creation is discussed in Section 4.2.3.1.

The path condition must be copied on Line 9 before applying the *must combine* mass production on the next line, such that the creation of the *may combine* mass production is not affected.

Then, we create the power-set out of the rules in the *may combine* rule set on Line 12. Note that this function would handle rule multiplicity and overlapping rules as in Section 4.2.3.2.

The last step is to iterate through the combinations created in the power-set. For each combination, a mass production is created and applied to the path condition. This is discussed in Section 4.2.3.3.

Note that the path condition itself is not modified, but instead a copy is made before application of the production. This is because each *may combine* mass production must apply over the same path condition, which was created by the application of the *must combine* mass production on Line 10. Each of these new path conditions is placed in a set on Line 17, which is returned as the output of the function on Line 18.

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.

4.2.3 Rule Combination

Definitions 21 and 22 create the sets of rules to be combined with that path condition, defining whether the rule may or must combine 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 present in the layer 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's executions are permitted.

Note that the following figures will depend on the matcher and rewriter for an example rule rl. The rule rl is shown in Figure 4.14a, the matcher $\lceil rl \rceil$ in Figure 4.14b, and the rewriter in Figure 4.14c.

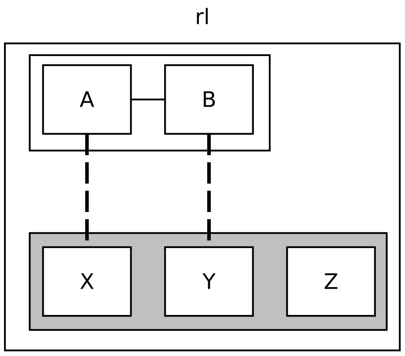
4.2.3.1 Must Combine Set

The *must combine* possibility represents the cases where the rule *rl* must match 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.

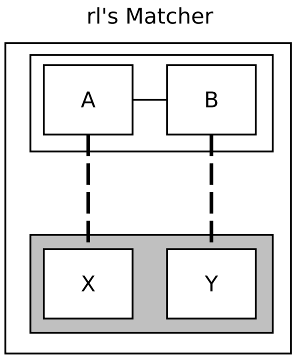
The elements from the rule *rl* will be added into the path condition *pc* using the rewriter in Figure 4.14c. Note that by definition of the *must combine* set, the rule does not contain negative elements.

An example of the rule being added to the path condition is seen in both subfigures in Figure 4.15. Note that this addition process is based on the double-pushout approach, such that new elements are created connected to where the rule's backward links overlap with the path condition's elements [50].

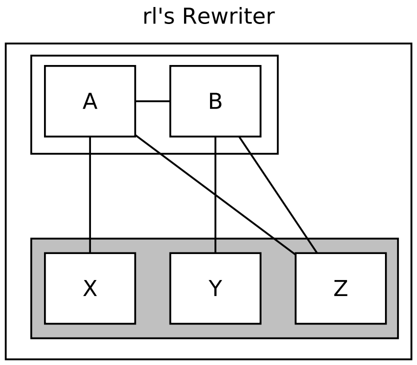
As well, multiple matches of the rule on the path condition will cause multiple additions as seen in Figure 4.15b.



(a) The rule *rl*.

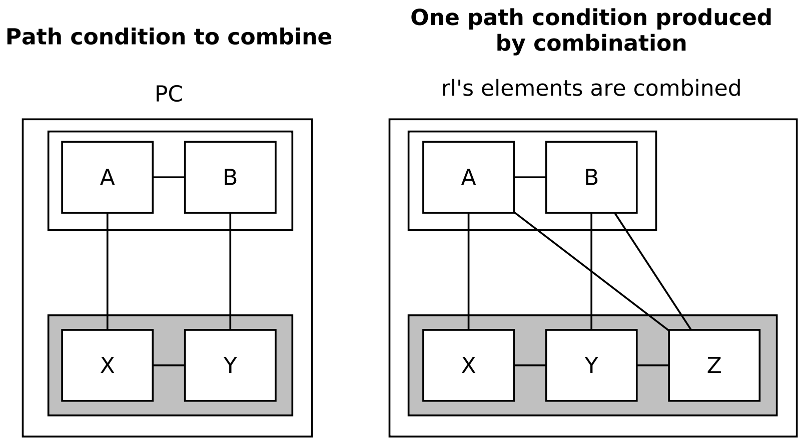


(b) The matcher for the rule.

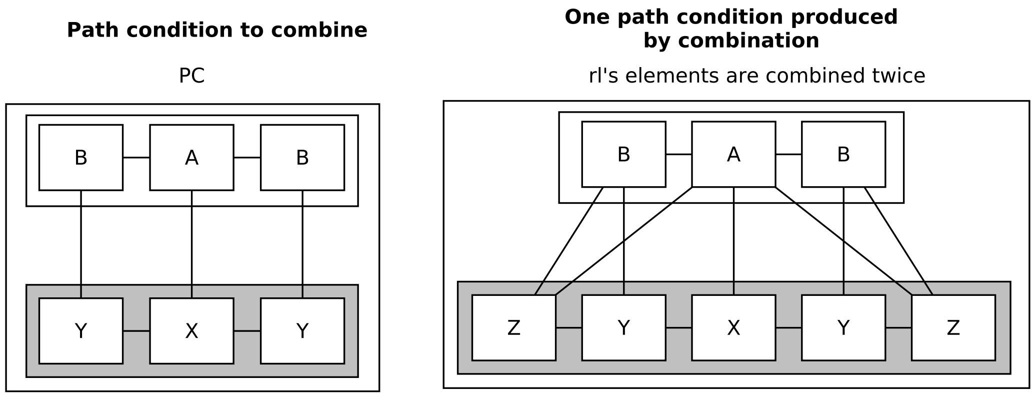


(c) The rewriter for the rule.

Figure 4.14 – The matcher and rewriter structures created for a rule used in building path conditions.


Path condition to combine
PC
A
B
X
Y
One path condition produced by combination
rl's elements are combined
A
B
X
Y
Z

(a) Satisfied at one location.


Path condition to combine
PC
B
A
B
Y
X
Y
One path condition produced by combination
rl's elements are combined twice
B
A
B
Z
Y
X
Y
Z

(b) Satisfied at multiple locations.

Figure 4.15 – Rule *r*'s dependencies are satisfied by path condition *pc*'s elements.

Must Combine Mass Production. We create a mass production p^*_{must} from the rule set $must_{pc}$ following the definitions in Section 3.2.5.1 and Ehrig *et al.* [50]. Note that it is permissible in DSLTrans to combine various rules together into one mass production as DSLTrans rules are confluent, as discussed in Section 3.4.

In this case, the production p^*_{must} is composed of the productions of all the rules in the $must_{pc}$ set, with each rule represented as many times as it matches over the path condition *pc*. As an example, the rule executes once in Figure 4.15a and twice in Figure 4.15b.

This mass production is then applied to the path condition *pc* in Section 4.2.2.5. This will combine the path condition with all rules in the layer that must apply due to the elements in the path condition.

Note that this application is performed *after* the mass production for the set of *may combine* rules has been created, but before that mass production is applied, as stated in Algorithm 1 on page 109. This is done such that the elements produced by the application of the *must mass production* are not considered by the construction of the *may mass production*.

4.2.3.2 May Combine

Once all the rules which *must* execute on the path condition have been gathered into a path condition, the rules which *may* combine must be considered. This set of rules, denoted may_{pc}, is slightly more complicated to handle. This is because there is a possibility that each of the rules in this set does not apply on the input-output models abstracted by that path condition.

As seen in the *must combine* 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 power-set of rule combinations from may_{pc}.

However, before we can generate all the *may combine* mass productions to be applied on the path condition, we must handle two cases in the power-set of rules in may_{pc}.

First, we must determine if we need to add extra rule applications to satisfy dependencies on the multiplicity of rule application, and second, we must examine whether the application of one rule subsumes the application of another. These cases are discussed in the following sections, along with a discussion on rules containing negative elements.

Multiple Rules. 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 the *may combine* 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, where it is also used to detect invalid rules and contracts as discussed in Section 6.4.1. Each element and link in the contract and rules are 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 power-set 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 power-set 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 we must investigate rule combinations concerning these rules.

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 apply on input models where rule A also apply as rule A has a subset of the element of rule B. Therefore, we can remove rule combinations from the power-set where rule B applies but rule A does not.

Note that detecting subsumption and removing the invalid combinations is not an optimization. It is instead a necessary operation for correctness, as it reflects the valid combinations during DSLTrans transformation execution.

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 may_{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 apply. As such, these rules are not placed in the rule set $must_{pc}$ as they are not guaranteed to apply.

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 apply. The abstraction relation (as discussed in Section 4.1) 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 4.16 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*.

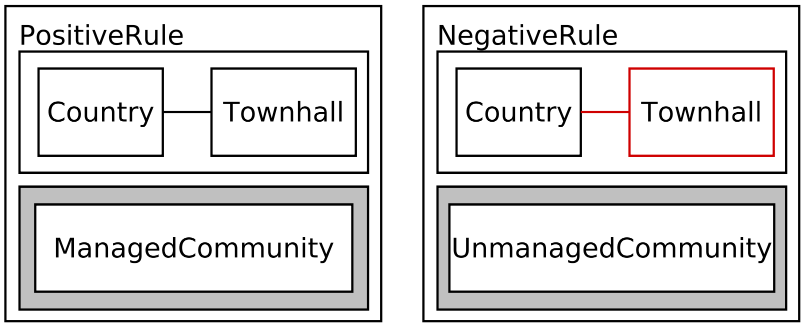


Figure 4.16 – Two rules which may execute on the same input graph.

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 apply.

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

4.2.3.3 Creating the May Combine Mass Production Sets

Once the power-set has been examined for possible rule overlap and insufficient rule multiplicity, we construct the set of *may combine* mass productions p^*_{may} through the application of mass production construction on each rule combination in the power-set.

Recall that a mass production is simply composed of the matchers and elements to be written for the rules in this set. The matcher for the *may combine* 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.

The elements to be written will be the same as in the *must combine* combination case. That is, the graph to be produced will be the rule's elements, along with traceability links added as discussed in Section 3.1.4.2. However, unlike the rewriter for rules during the execution of a DSLTrans transformation, the structures for adding elements to path conditions may contain negative elements.

4.2.4 Considering Further Layers

Section 4.2.3 and Algorithm 1 describe 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 4.1, this set of path conditions will represent every feasible transformation execution. Section 4.4 discusses our approach to proving contracts on these path conditions, and thus on all executions of the transformation.

4.3 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 4.4. In particular, these properties ensure that our building of path conditions is proper, that the construction reflects the semantics of DSLTrans as discussed in Section 3.2, and that the contract proving approach in Section 4.4 is valid.

4.3.1 Representation

Proposition 3. *Every path condition abstracts at least one valid transformation execution.*

Proof. Let $tr \in \textsc{Transforms}$ be a DSLTrans transformation. We wish to demonstrate that for all path conditions, there exists a transformation execution (represented by an input-output model) that the path condition abstracts through the abstraction relation. That is: $\forall pc \in \textsc{Pathconds} : (\exists iom \in \textsc{IOMs} \mid pc \mapsto iom)$

Note that this abstraction is bound by the restriction that $getTransformationPC(pc) = tr = getTransformationIOM(iom)$. That is, we are only concerned with path conditions and input-output models for the same transformation.

This representation property will be proved by performing induction on the number of rules represented by a path condition *pc*. That is, rules will be added one-by-one to the rule set abstracted by the path condition. This explanation will follow the pattern of construction in Section 4.2, and the DSLTrans semantics presented in Section 3.2.

4.3.1.1 Base case:

The base case is when *pc* is the empty path condition ϵ_{pc}. That is, the set of rules represented by the path condition is empty. Section 4.1.5.1 demonstrates that the empty path condition abstracts any empty input-output model, as well as any input-output model for which no rule applies.

This is due to no input elements or traceability links in the path condition missing in the input-output model, and no output elements or traceability links in the input-output model missing in the path condition.

4.3.1.2 Inductive case:

Assume that a path condition *pc* abstracts at least one input-output model. Then the inductive step adds a rule *r* to the rule set which *pc* represents, resulting in a path condition pc'. The representation property will be shown to also hold for this new path condition pc'.

This proof will depend on how the rule rl combined with the original path condition pc. There are three cases of rule combination to consider, 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 holds for Case 1 as the path condition pc' is not changed from pc and therefore the original input-output models are still abstracted.

When Cases 2 or 3 occur, new elements are added to the path condition pc to form pc'. Both cases are based on combining a path condition with a rule, as laid out in Section 4.2.2.

Adding a Rule. We start by picking an input-output model iom such that the original path condition pc abstracts iom, which exists by our induction hypothesis. Another input-output model iom' is then built by adding input elements such that the rule rl executes. Let us now demonstrate iom' is abstracted by the path condition pc', where pc' is formed by combining the rule rl with pc.

Below we examine how the addition of the rule affects the conditions of the abstraction relation explained in Section 4.1.4.1:

1. a semi-injective morphism must exist between the input elements of the rule components in the path condition and the input of the input-output model
2. a surjective morphism must exist between the output of the input-output model and the output part of the path condition
3. a semi-injective morphism must exist between the traceability links of the path condition and the input-output model
4. a surjective morphism must exist between the traceability links in the input-output model and the path condition

5. the morphisms for the above conditions must be consistent with one another according to Equation 2.1 on page 41

Our arguments are based on the fact that if the morphisms hold in the above conditions before addition of the rule elements, then they will also hold after. Note that our arguments are applicable both when the rule *may* and *must* combine with the path condition. This is because both additions add the elements from the rule without removing any.

Condition 1. Let us start by arguing for why Condition 1 holds for the new path condition pc' and the new input-output model iom'.

The application of the rule rl in the input-output model iom' means that the elements present in the rule's matcher $\lceil rl \rceil$ were present in the input-output model. The rewriter of the rule $\lfloor rl \rfloor$ then creates the output elements of the rule in the input-output model.

For the path condition pc', these are exactly the elements added during path condition construction. Whether the rule *may* or *must* combine with the path condition, all rule elements (including added traceability links) are added during the application of the mass productions (Section 4.2.2.5). Therefore, there will continue to be an injective morphism found between the added elements in pc' and the added input elements in iom'.

Condition 2. A similar argument for Condition 1 is used for Condition 2. Traceability links will be created in the input-output model iom' when the rule rl applies, as they are created for the rule's rewriter (Section 3.1.4.2).

During path condition construction, these traceability links are explicitly added to the rule elements, before the elements are added to the original path condition pc (Section 4.2.2.5). Thus, it will be possible to make an injective match on the traceability links in the path condition pc' to the traceability links in the input-output model iom'.

Condition 3. The argument for Condition 3 follows the argument for Condition 1. As the application of the rule rl in iom' has produced output elements, these elements must be represented in the path condition pc' through a surjection relation. These elements exist, as the combination of pc' and rl adds all elements from the rule. Therefore, the same output elements created in iom' are also created in pc', and the surjection relation continues to hold.

We must also consider the repeated application of the rule in the input-output model iom'. For each application of the rule, output elements will be created in the output component of the input-output model iom'. However, the nature of the surjection relation allows multiple elements in the input-output model to match over the same element in that path condition, as seen in the examples in Section 4.1.5. Therefore, the surjection relation will continue to hold after the rule applies multiple times in the input-output model.

Condition 4. Condition 4 states that there must be a surjective morphism from the traceability links in the input-output model iom' to the traceability links in the path condition pc'. As with Condition 3, the traceability links in the input-output model are created when the rule applies in the transformation. These applications are directly represented in the path condition when the rule (with traceability links added) is combined with the path condition.

As well, the surjective nature of the morphism means that repeated rule application in the input-output model will match over fewer representations in the path condition. Thus, a surjective match can be found.

Condition 5. Finally, Condition 5 refers to the need for all morphisms in the abstraction relation to have consistent mappings. This consistency is defined in Equation 2.1 on page 41, where all mappings originating from a source element must map onto the same target element in the morphisms from the other conditions.

We assume that the morphisms are consistent before the addition of the rule to both the path condition and the input-output model. The rule with the same elements is then added to both sides. Therefore, the consistency of the morphisms is retained.

□

4.3.2 Coverage

Proposition 4. *(Coverage) Every input-output model is abstracted by at least one path condition.*

Proof. Let $tr \in \textsc{Transf}$ be a DSLTans transformation. We wish to demonstrate that, for all input-output models *iom* for the transformation, there exists a path condition *pc* such that *iom* is abstracted by *pc*. That is:

$$\forall iom \in \text{IOM}\textsc{s} : (\exists pc \in \textsc{PathConds} \mid pc \mapsto iom)$$

Note that this proof involves the claim that the abstraction relation is sufficiently powerful enough to allow for a path condition to abstract the input-output model. This claim is based on our abstraction relation's definition being intertwined with the building of each path condition, such that symbolically executing a rule in a path condition directly mirrors the application of a rule in a input-output model. This replication of DSLTrans' semantics (Section 3.2) mirrors Proposition 3, where a path condition will always abstract at least one input-output model. Therefore, our claim is that no input-output model can be created that is not abstracted by a path condition.

First, to discuss coverage let us examine the set of input-output models where no rules apply. These input-output models can contain either an empty input model, or an input model such that no rule matches. As discussed in Section 4.1.5.1, these input-output models are represented by the empty path condition ϵ_{pc} and are therefore covered.

In the rest of this proof we are thus considering only those input-output models where a set of transformation rules have applied. To assert that these input-output models are covered by at least one path condition, we will examine the possibility of a counter-example. Therefore, we provide a proof by contradiction.

Assume there exist two input-output models which represents two sets of rules which differ by the application of a rule rl. One input-output model is abstracted by some path condition, while the input-output model of interest iom is abstracted by no path conditions. To show that iom cannot exist, we examine how the application of the rule rl could have led to a failure of the abstraction relation:

The rule was not symbolically executed during path condition construction. The first possibility of not building an appropriate path condition for iom goes to the heart of our technique, where we consider the case that the application of rl has not been represented on the path conditions created. Note that during path condition construction (Section 4.2) that our technique is not conservative about building path conditions. This is required as the set of path conditions must cover every possible input-output model.

In particular, we examine every rule to determine if there is a possibility of symbolic execution or not. We only consider a rule rl to not symbolically execute on a path condition pc if the rule enforces dependencies which are not found in the path condition. That is, $rl \in not_{pc}$. This is correct as these dependencies must be satisfied for the rule to apply, which mirrors the presence of the elements in the abstracted input-output models.

In all other cases, the rule is considered to potentially (or definitely) apply, and the rule is placed in may_{pc} or $must_{pc}$. In this case, multiple path conditions are generated with that rule symbolically applied. Therefore, if a rule has not been applied on a path condition, then that rule cannot have applied in an input-output model.

The rule was not symbolically executed the appropriate number of times. The second possibility for no path condition abstracting an input-output model is that the multiplicity of the rule application in the input-output model has not been properly considered. This case may arise sure to dependencies between rules, such that one rule may need to be applied multiple times for another rule to be applied. This is discussed in Section 4.2.3.2 on page 115

In our symbolic execution technique, we rely on the verification user to explicitly define how many times each rule should symbolically apply. This information can be obtained with our rule dependency analysis as discussed in Section 6.4.1 on page 238. Therefore, we assume that all rules have been duplicated an appropriate number of times to create suitable path conditions such that rule multiplicity does not cause abstraction issues.

Elements are present in the *Input* graph of all path conditions, but are not present in the *Input* graph of the input-output model. For the third case, where no path condition represents the input-output model, we consider a possibility where all path conditions have too many elements in their input graph. That is, the input-output model has strictly fewer elements in its input graph than all path conditions. This case cannot arise due to the iterative approach to build up the set of path conditions. In this iterative approach, a path condition is created for each possibility when rules may or may not apply (Section 4.2.3.2).

Therefore, there will always exist a path condition with a subset of those elements present in the input-output model. In the base case, the empty path condition ϵ_{pc} will represent the input-output models with no input elements.

Elements are present in the *Output* graph of the input-output model, but are not present in the *Output* graph for any path condition. The fourth case regards the output elements in the input-output model. Due to the out-place semantics of DSLTrans, these elements must have been created through the application of rules.

Therefore, as long as there exists a path condition which captures the application of those rules, then all output elements will be properly represented by that path condition. This case thus also depends on the correct building of all possible transformation executions taking into account rule multiplicity and dependencies.

Given that all executions are represented, then the output elements will be represented in at least one path condition.

□

4.4 Contract Proof

The algorithm presented in Section 4.2 produces all possible path conditions for a particular DSLTrans transformation, representing all possible input-output models. This section will detail another contribution of this thesis: a method to prove structural contracts on the transformation by creating a set of path conditions where the contract does hold, and a set where the contract does not hold.

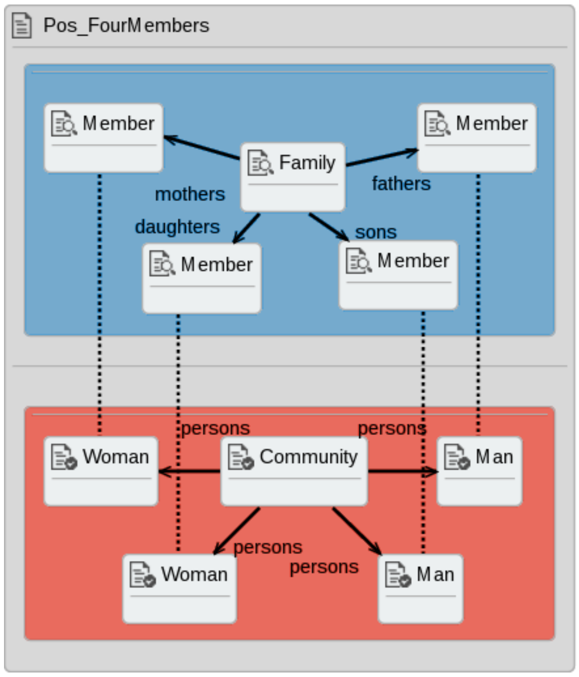
The abstraction relation presented in Section 4.1 can then be used to extend the contract proof result to the infinite set of input-output models for the transformation. A positive result for contract proof means that the contract will hold on all abstracted input-output models, while a negative result means that the contract will fail on at least one abstracted input-output model. This is further explored in Section 4.5.

4.4.1 Structure of a Contract

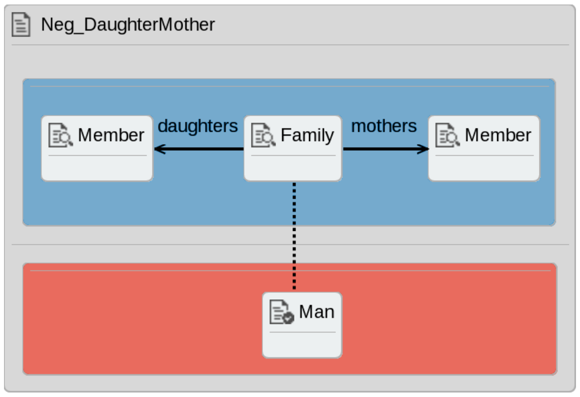
Contracts intentionally use a similar structure to DSLTrans rules, input-output models, and path conditions, as detailed in Definition 24. However, in contracts we allow the possibility of using indirect links in both the input and output graphs. As well, we must forbid the use of negative elements in contracts. As explained in Section 4.5.1.1, this is due to the abstraction relation allowing a path condition to represent a lower bound for abstracted input-output models.

Figure 4.17 shows two examples of contracts for the *Families-to-Persons* transformation. As with structures like path conditions and input-output models, the top half of a contract is the *Input* graph, while the bottom half is the *Output* graph. Two types of backward links connect elements in the *Input* and *Output* graphs, and are used to define dependencies on element creation.

Contracts form a pre-condition/ post-condition structure to be proven on path conditions. When the elements in the *Input* graph (and elements in the *Output*



(a) *Pos_ FourMembers* contract – No counter-examples found.



(b) *Neg_ DaughterMother* contract – Counter-examples found

Figure 4.17 – Contracts proved on the *Families-to-Persons* transformation.

graph connected by *pre* backward links) matches over a path condition, then the entire contract structure must match over that path condition for the contract to be satisfied. Otherwise, the path condition becomes a counter-example to the contract. This is further discussed in Section 4.4.2 where we precisely define satisfaction of a contract.

Definition 24. *Contract*
A contract c is a five-tuple
$\big\langle Input,\ Output,\ back_{pre},\ back_{post},\ (s_{back}, t_{back}),\ E_{indirect},\ \big\rangle$, *where:*

— $Input, Output \in$ TG

— *Input and Output may be empty and are disjoint*

— $back_{pre}$ *and* $back_{post}$ *are two sets of 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* $s_{back} : E_{back_{pre}} \cup E_{back_{post}} \rightarrow V_{Output}$ *and* $t_{back} : E_{back_{pre}} \cup E_{back_{post}} \rightarrow V_{Input}$ *that respectively provide the source and target vertices for each backward link*

— $E_{Indirect} \in E_{Input}$ *is the set of indirect links.*

The set of all contracts for a transformation *tr* is termed $\textsc{Contracts}_{tr}$.

We also define a utility function *getTransformationContract*: $\textsc{Contract} \rightarrow \textsc{Transforms}$. 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.

Backward Link Types. Note that in a contract we have two types of backward links, which are used to check for the creation of elements in the path condition. The first type of backward link, termed *pre* backward links and stored in $E_{back_{pre}}$, are those backward links that must be present in the pre-condition for the contract to hold. The second type of backward links, termed *post* backward links and stored in $E_{back_{post}}$ will be present in the post-condition of the contract. The two contracts in Figure 4.17 (and throughout this thesis) contain the *post* type of backward links. Figure 4.18 shows the visual syntax for the two backward link types.

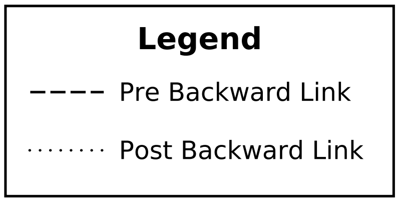


Figure 4.18 – Visual syntax for *pre* versus *post* backward links in contracts.

To illustrate the difference in the two types of backward links, we present in Figure 4.19 a graphical representation of the *ChannelProduction* contract from Section 7.1 on page 251. Note that the backward link present in the contract is a *post* backward link. Therefore, the contract represents the informal statement, 'If there exists a *Channel* element in the input model, then an *ATOM* element should be produced in the output model, connected to two other elements'. Note how the existence of the *ATOM* element in the output model is only checked in the post-condition of this contract.

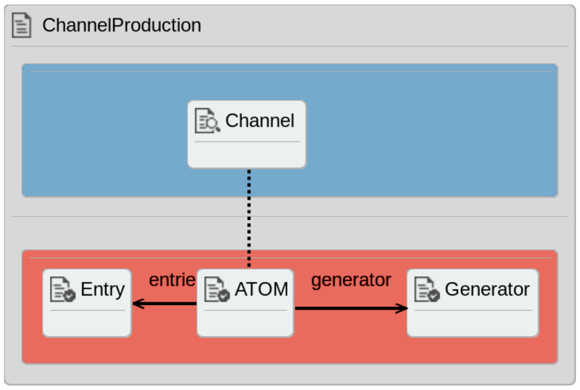


Figure 4.19 – The *Channel Production* contract for the *RSS-to-ATOM* transformation.

In contrast, if the backward link in Figure 4.19 was a *pre* backward link, the informal statement would be 'If there exists a *Channel* element in the input model which has produced an *ATOM* element in the output model, then that *ATOM* element should be connected to two other elements'. Therefore, the use of the two backward link types allows the contract to pre-suppose the application of rules within the transformation, and gives the contract language further expressibility. Other contract constructs are discussed in Section 4.6.

4.4.2 Contract Satisfaction

The section will detail the conditions necessary for an input-output model and path condition to *satisfy* a contract. This satisfaction is determined by matching the contract's pre-condition matcher, and potentially the contract's post-condition matcher on the input-output model and path condition.

This section begins by discussing how to build the contract's pre-condition and post-condition matchers. Then, we examine how a contract is satisfied by an input-output model and path condition in Section 4.4.2.2.

Result	**Symbol**	**Explanation**
Not Applicable	$\vert\!\rhd$	The contract's pre-condition does not match on the input-output model/ path condition.
Does Not Satisfy	$\not>$	The contract's pre-condition does match, but the contract's post-condition does not match.
Satisfies	$>\!\vert$	The contract's pre-condition and post-condition both match.

Table 4.1 – Results for contract proof on an input-output model or path condition.

Table 4.1 shows the three results for satisfaction, along with the symbols which represent these results. For example, a path condition *pc* not satisfying a contract *c* would be written $pc \not> c$. These three results will be used in Section 4.4.3 to divide the set of path conditions for the transformation, and in Section 4.5 to discuss how the result of contract proof on a path condition represents the result of contract proof on the input-output models abstracted by that path condition.

4.4.2.1 Contract Matchers

The structures for the contract's pre-condition and post-condition are created from the contract structure defined in Definition 24. Recall that a contract c is a five-tuple $\big\langle Input,\ Output,\ back_{pre},\ back_{post},\ (s_{back}, t_{back}),\ E_{indirect}\big\rangle$.

Definition 25. *Contract Pre-condition Matcher*
We define the contract c's pre-condition matcher, noted $\lceil c \rceil_{pre}$*, to be a typed graph four-tuple* $\langle V^{pre}, E^{pre}, (s^{pre}, t^{pre}), \tau, E^{pre}_{indirect}\rangle$*, where:*

— $V^{pre} = V_{Input} \cup \big\{v \in V_{Output} \mid \{\exists e \in E_{back_{pre}} | s_{back}(e) = v\}\big\}$

— $E^{pre} = E_{Input} \cup E_{back_{pre}}$

— $s^{pre} = s_{Input} \cup s_{back}$, $t^{pre} = t_{Input} \cup t_{back}$

— $\tau = \tau_{Input} \cup \tau_{Output}$

— $E^{pre}_{indirect} = E_{indirect} \cap E^{pre}$

The pre-condition for the contract is created in Definition 25. It consists of the elements in the *Input* graph of the contract, as well as the backward links marked *pre* and *Output* elements attached to those *pre* backward links. As well, the set of indirect links in the pre-condition is restricted to those edges retained.

Definition 26. *Contract Post-condition Matcher*
We define c's post-condition matcher, noted $\lceil c \rceil_{post}$*, to be a typed graph four-tuple* $\langle V^{post}, E^{post}, (s^{post}, t^{post}), \tau, E^{post}_{indirect}\rangle$*, where:*

— $V^{post} = V_{Input} \cup V_{Output}$

— $E^{post} = E_{Input} \cup E_{Output} \cup E_{back_{post}}$

— $s^{post} = s_{Input} \cup s_{Output} \cup s_{back}$, $t^{post} = t_{Input} \cup t_{Output} \cup t_{back}$

— $\tau = \tau_{Input} \cup \tau_{Output}$

— $E^{post}_{indirect} = E_{indirect}$

The post-condition for the contract is created in Definition 26. It consists of all *Input* and *Output* elements from the contract, along with all backward links marked as *post*.

4.4.2.2 Input-Output Model Satisfaction

Given the matchers for the contract defined in Definitions 25 and 26, we detail here how an input-output model satisfies a contract. Due to the common structure between contracts, path conditions, and input-output model, this satisfaction is based on whether the contract's matchers can be found in the input-output model using a morphism.

Definition 27. *Contract Satisfaction by an Input-Output Model*
Let c be a contract in $\textsc{Contracts}(tr)$ *and iom be an input-output model in IOMs*(tr) *for a transformation tr.*

The input-output model iom satisfies the contract c, which is written iom $\gtrdot$ *c, iff* $\exists f, g$ *morphisms such that:*

$$(\lceil c \rceil_{pre} \underset{f}{\rightarrow} iom \wedge \lceil c \rceil_{post} \underset{g}{\rightarrow} iom) \wedge (\forall v \in V, e \in E : f(v) = g(v), f(e) = g(e))$$

The input-output model iom is not applicable for the contract c, written iom $\not\vartriangleright$ *c, iff* $\nexists f$*. As well, the input-output model iom does not satisfy the contract c, written iom* $\not\gtrdot c$*, iff* $\exists f$ *but* $\nexists g$*.*

Definition 27 states that an input-output model satisfies the contract if the contract's pre-condition matcher and post-condition matcher can both be found in the input-output model through two morphisms. Note that these morphisms must also be consistent, such that the same element in the matcher must match to the same element in the input-output model for both morphisms.

As well, the input-output model is said to not satisfy the contract if the f morphism is found for the contract's pre-condition matcher, but the g morphism cannot be found for the contract's post-condition matcher.

4.4.2.3 Path Condition Satisfaction

As there exists an infinite number of possible input-output models for the transformation, it would be impossible to prove contracts by enumerating these input-output models. Our technique thus relies on the finite set of path conditions created for the transformation, such that we prove contracts on these path conditions and extend the results to all input-output models.

Definition 28. *Contract Satisfaction for a Path Condition*

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

Path condition pc satisfies contract c, written $pc \mathrel{>\!\!|} c$, iff $\exists f, g$ typed graph split morphisms such that:

$$(\lceil c \rceil_{pre} \underset{f}{\rightarrowtail} pc \wedge \lceil c \rceil_{post} \underset{g}{\rightarrowtail} pc) \wedge (\forall v \in V, e \in E : f(v) = g(v), f(e) = g(e))$$

The path condition pc is not applicable for the contract c, written $pc \not\triangleright c$, iff $\nexists f$. As well, the path condition pc does not satisfy the contract c, written $pc \not> c$, iff $\exists f$ but $\nexists g$.

The principle behind the satisfaction relation in Definition 28 is similar to the satisfaction relation between a contract and an input-output model in Definition 27. 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 the morphisms are different in Definition 27 and Definition 28. When checking for satisfaction of a contract by an input-output model, the morphisms f and g are isomorphisms as defined in Definition 4. This forces the elements in the contract to be found with their structure in the input-output model. However, when checking for satisfaction of a contract by a path condition, we instead use a typed graph split morphism, as defined in Definition 5 on page 37. This allows for the elements in the contract to "split" over multiple elements in the path condition.

4.4.2.4 Satisfaction Example

For example, Figure 4.20 shows how the pre-condition of the *Neg_-DaughterMother* contract is matched over an input-output model and a path condition. The backward link in the *Neg_DaughterMother* contract is a *post* backward link. Therefore, the contract's pre-condition matcher consists of only the *Family* element, the two *Member* elements, and the *mothers* and *daughters* associations.

Figure 4.20a shows how the contract is satisfied by an input-output model. The elements and associations in the contract's pre-condition matcher are isomorphically found in the input-output model.

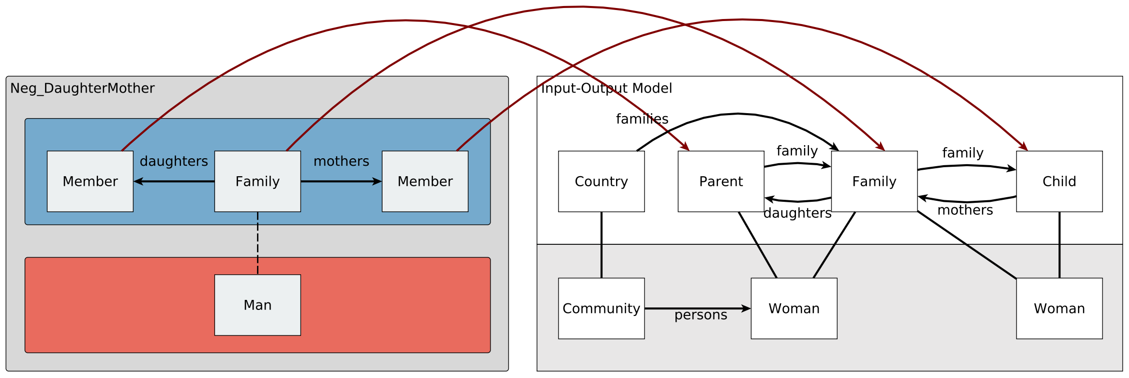
In contrast, Figure 4.20b demonstrates the matching process of the contract's pre-condition matcher on a path condition using the typed graph split morphism. Note how the edges in the contract are matched injectively to the path condition's edges, but the vertices in the path condition are matched to two vertices in the input-output model. This allows the *Family* element in the contract's pre-condition matcher to 'split' over the two *Family* elements in the path condition.

As seen in Section 4.1.5, there is an abstraction relation between the input-output model in Figure 4.20a and the path condition in Figure 4.20b. In Section 4.5, we shall examine how all these morphisms commute to allow the result of proving a contract on a path condition to extend to the abstracted input-output models.

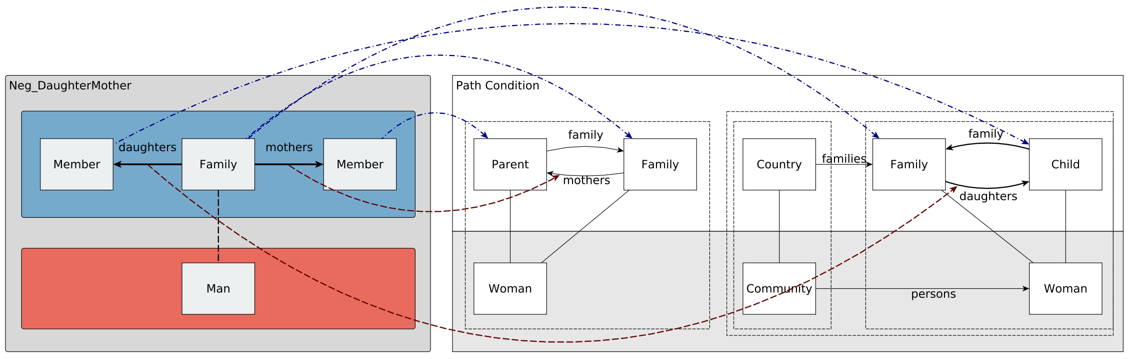
4.4.3 Transformation Verification

Section 4.2 describes how the set of path conditions PathConds(tr) is built for a transformation tr. With the definitions from the last section of how a path condition satisfies a contract, we can then build two sets of results for each contract c in Equations 4.7 and 4.8.

The first set sat_c is those path conditions which satisfy the contract, while the second set $nonsat_c$ is those path conditions that do not satisfy the contract.



(a) Matching over an input-output model using an injective morphism.



(b) Matching over a path condition using the typed graph split morphism.

Figure 4.20 – Two different morphisms used when matching the *Neg_-DaughterMother* contract.

These sets are returned from the contract proof algorithm as the result, and their examination gives insight into the interactions of transformation rules.

$$sat_c = \{\forall pc \in \textsc{PathConds}(tr) \mid pc \rightarrowtail c\} \tag{4.7}$$

$$nonsat_c = \{\forall pc \in \textsc{PathConds}(tr) \mid pc \not\rightarrowtail c\} \tag{4.8}$$

Note that a third set $notappl_c$ of path conditions could be created out of those path conditions that are not applicable for the contract. However, this set is not relevant for verification by definition.

Additional Verification. Note that in our current implementation of the contract prover, we have developed initial support for providing detailed information to the user at the end of contract verification, as presented in Section 6.4.3. These analyses allow the user to precisely determine in which cases the contract holds, and in which cases it does not hold, such that they have assurance that the result is as intended.

As well, we also perform static checks on the transformation and the contract. These analyses are discussed in Section 6.4, and involve determining whether 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 which indicates an invalid 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 case studies (Chapter 7).

4.5 Validity of Contract Proof

Section 4.3 discusses the *representation* and *coverage* properties of the path condition building technique, while Section 4.4 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 5.

Proposition 5. *(Validity) The result of proving a contract on a path condition represents the result of proving the contract on all input-output models abstracted by that path condition.*

Recall from Section 4.1 that $pc \mapsto iom$ *means that the path condition* pc *abstracts the input-output model* iom*.*

Let $tr \in$ TRANSFORMS be a transformation and $c \in$ CONTRACTS(tr) be a contract for tr. Then, $\forall pc \in$ PATHCONDS(tr), the following three implications must be satisfied.

4.5.1 Implication 1 - Positive Satisfaction

$$\forall iom \in IOM_{tr} : (pc \mapsto iom \wedge pc \succ\!\!\mid c) \implies iom \mid\!\!\rhd c \vee iom \succ\!\!\mid c \tag{4.9}$$

Equation 4.9 states the first implication, which is that if the path condition satisfies the contract, then all abstracted input-output models must either be not applicable to the contract, or they must satisfy the contract.

This can be proven by examining the commutativity of the contract proof morphisms, which determine how path conditions and input-output models satisfy the contract (Section 4.4.2.2 and 4.4.2.3).

Element Existence. The key insight to our argument is that the path condition represents all the relevant elements present in those input-output models abstracted through the abstraction relation (Section 4.1). The examples in Section 4.1.5 show the representation between the elements in the path condition and in the input-output model. In particular, the path condition will represent the application of the same rules as the input-output model as was argued in Section 4.3.

The contract proof morphisms are checking whether the elements in the contract are present in the path condition. Therefore, by composing the morphism between the contract and the path condition, and the path condition and the input-output model, we know that the elements are present in the input-output model. These morphisms are shown for a concrete example in Figure 4.11 and Figure 4.20b.

Element Overlapping. Both the input and output elements in the input-output model are guaranteed to be in the path condition through the abstraction relation. However, there is an issue of determining whether the composition of morphisms retains the structure required.

Note that the split morphism is used in matching the contract over the path condition (Definition 28). This morphism allows one element in the contract to match over two elements in the path condition. As well, the morphism used to match path conditions over input-output models (Definition 20) allows two elements from different rules in the *Input* graph of the path condition to match over the same element in the input-output model.

This splitting and joining of elements from composing these morphisms means that the exact structure present in the *Input* graph of the path condition may not be present in the input-output model. As the matchers of the contract require an injective matching (Definition 27), this means that there is a possibility that the contract's pre-condition matcher may not match on the input-output model, even if the matcher did match on the path condition. However, this is acceptable, as this is the definition of the input-output model being *not applicable* to the contract.

If the pre-condition matcher of the contract matches over the input-output model, then the issue is whether the path condition has accurately represented the elements present in the input-output model's *Output* graph. The abstraction relation indicates that the elements present in the *Output* graph of the path condition must be a subset of the elements present in the input-output model, through a surjection relation from the input-output model to the path condition. Thus, if the post-condition matcher of the contract is satisfied by the path condition's *Output* elements, it will also be satisfied by the abstracted input-output model *Output* elements.

4.5.1.1 Negative Contract Elements

As discussed in Section 4.1, the abstraction relation allows for more input elements in the input-output model 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 input-output model.

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 input-output model 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. However, the semantics of this operator means that we are verifying that the elements are not *always* present in all abstracted input-output models, as further explained in Section 4.6.2.

4.5.2 Implication 2 - Negative Satisfaction from Path Condition

Equation 4.10 states the second implication where if the contract fails to hold on a path condition, then the path condition abstracts an input-output model where that contract will also fail to hold. Thus the path condition will serve as a counter-example to the contract.

$$pc \not\models c \implies \big(\exists iom \in IOM_{tr} \mid pc \mapsto iom \wedge iom \not\models c\big) \tag{4.10}$$

The input-output model *iom* which fails the contract can be easily constructed to be a duplicate of the path condition *pc*. That is, the input model of the input-output model will be composed of the elements present in the input model of the path condition such that the rules applied and the output elements will be the same in both. This is allowed by the abstraction relation, as elements in different rules are not required to overlap with each other.

In other words, contract satisfaction is specified as a morphism for both satisfaction on input-output models (Definition 27) and for path conditions (Definition 28). For input-output models, this is an injective morphism, while for path conditions the contract's matchers are matched by the typed graph split morphism. This morphism is unique, but has the advantage that it can also act as an isomorphism. Thus, if an input-output model is created as a copy of the path condition, the contract's matchers will fail to match on the input-output model as well, generating the counter-example.

4.5.3 Implication 3 - Negative Satisfaction from an Input-Output Model

Finally, Equation 4.11 states the third implication, where if the contract fails to hold on a input-output model, 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 input-output models abstracted.

$$\forall iom \in IOM_{tr} : (iom \not\models c \land pc \mapsto iom) \implies pc \not\models c \tag{4.11}$$

If the contract fails to be satisfied by an input-output model, then the contract's pre-condition matcher has matched over the input-output model, but the contract's post-condition has failed to match. We will argue how the path condition reflects these matching results for the contract's pre-condition matcher, and the post-condition matcher.

Pre-Condition Matcher. The abstraction relation allows for the *Input* graph of the input-output model to contain more elements than the input graph of the path condition (Section 4.1.4.1). Therefore, it is possible that the pre-condition matcher would match over the input-output model and not the path condition. However, this failure of representation is solely due to our abstraction over the multiplicity of rule application. As mentioned in Section 4.5.4, it is entirely possible that the path conditions created have not applied a rule enough times to satisfy the contract. As well, the contract may be matching over an element which is not present in any rule in the transformation.

We handle these multiplicity and out-of-transformation concerns in a step prior to path condition construction. A number of checks (detailed in Section 6.4) ensures that rules are set to be symbolically applied the correct number of times, and that all required contract elements are present in the transformation. Failure of these checks indicates a situation where the user is required to intervene and fix the transformation or contract.

During contract proving, we require that these concerns have been addressed. Therefore, a path condition will exist with the proper rule multiplicity, that path condition will abstract the input-output model in question, and the contract's pre-condition matcher will match on the path condition.

Post-Condition Matcher. If the contract's post-condition matcher fails to match on the input-output model, then this means there is an issue with finding the traceability links or *Output* elements in the input-output model. The multiplicity assumption defined above allows us to state that the number of traceability links in the path condition will be a sufficient lower bound to those in the input-output model. Thus, the issue is that the elements in the input-output model's *Output* graph could not be found.

If the output elements cannot be found in the input-output model, they will not be present in the path condition through the surjection relation in the abstraction relation. This surjection relation ensures that the *Output* elements in the input-output model are a superset of those found in the path condition. Thus the failure of the post-condition to match on the input-output model is reflected in the failure to match on the path condition.

An example of this case can be seen in Figure 4.11 on page 102 and Figure 4.20b on page 135. Note that the contract's post-condition contains a *Man* element which is not found in either the input-output model or the path condition. The abstraction relation ensures that the *Output* elements in both structures consists of solely *Woman* elements, and therefore the contract will not be satisfied by either structure.

4.5.4 Partitioning of Transformation Executions

As explained in Section 4.2.3.2, our abstraction of rule multiplicities implies a partitioning of the input-output models by path conditions. In fact, path conditions can abstract over the same subset of input-output models, which has consequences for contract proving.

In particular, our contract proving technique must be careful about how this non-unique abstraction relation 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 input-output models that are abstracted by pc_{super} are a superset of those abstracted by pc_{sub} due to this abstraction over the multiplicity of A.

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 4.5.1.1. 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*, as elements can only be added.

Thus the case of interest is when pc_{super} does not satisfy c and pc_{sub} does satisfy 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 input-output models where the contract is satisfied. 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 4.2.3.2), 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.

Note that in our implementation of the contract prover, we provide an analysis on the dependencies of rules and contracts to ensure that each rule is applied enough times. This is discussed in Section 6.4.1.

4.6 Contract Expressiveness

This section discusses the three elements of the contract language. The first element is a small language for creating and matching attribute values. The second element is a contract itself comprised of a pre-condition/ post-condition pair, which is here termed an *atomic contract*. The third element is a set of propositional logic constructs which can be used to compose atomic contracts into larger contracts.

This section also presents example contracts using the case studies examined in Chapter 7. In particular, we present examples for attribute checking, the propositional logic constructs, and a preliminary classification of contracts by intention.

4.6.1 Attributes

As the DSLTrans language offers the ability to set (String) attribute values for the output elements, the contract language has constructs to construct and match these values. Modelled on the constructs available in DSLTrans (Section 2.2.5 on page 30), the contract language can refer to attribute values, concatenate values, and employ atoms (literals).

For example, the *MotherFather* contract for the *Families-to-Persons* transformation in Figure 7.10 on page 265 contains attribute value checks in the post-condition. These checks determine if the full name of the produced *Person* elements has been correctly created from the last name of the *Family* and the first name of the *Member*.

However, there is a concern in how the matching of the contract's attribute-setting equations is performed over a path condition. Recall that path conditions represent the application of a set of rules on an arbitrary input model. Therefore, there are no concrete values given to the *firstName* or *lastName* elements in the path condition when the contract is matched.

As described in Section 3.2.3 on page 59 and touched upon in Section 4.1.6, we rely on the presence of an attribute solver which is able to resolve whether or not the attribute equations match. Our current implementation of this solver is discussed in Section 6.2.2 on page 201.

In the case of the *MotherFather* contract, the pattern equation is modelled as a concatenation of an attribute reference to the *firstName* value and a reference to the *lastName* value. The equation which is placed on the *Member* elements by the symbolic execution of the *Father2Man* or *Mother2Woman* rules is similar. Therefore, the solver must be able to parse these equations and determine that they both contain the same attribute references.

Note that to allow for more flexibility with the value of an attribute, a *Wildcard* element also exists within the contract language. This *Wildcard* element is demonstrated in the *AssignmentInstance* contract (Figure 7.16) for the *mbeddr* transformation, where it is used to check whether the name of various elements ends with particular strings. In our solver implementation, these wildcards are resolved with regular expressions which are able to match on a non-empty set of characters.

4.6.2 Propositional Logic and Pivots

The thesis of Selim [135] defined propositional contract logic constructs (*And*, *Or*, *If-Then*, and *Not*), along with pivots which can be used to enhance the expressiveness of the contract language. These constructs are heavily used in the GM-to-AUTOSAR and UML-to-Kiltera case studies (Sections 7.2 and 7.3).

Note that this thesis offers an extension over that earlier work, in that we have redefined the semantics of these constructs to include the explicit result of a contract's pre-condition not being found in a path condition, making the contract not applicable for that path condition. This allows for the easy division of path conditions into the success and failure sets given by the contract proving technique (Section 4.4.3).

As well, we clarify the semantics of the *Not* construct, which has unintuitive semantics. Recall that the success of a contract means that the contract's elements are always present in the path condition. Therefore, the negation of that contract succeeds when the elements are not *always* created.

Contract Satisfaction. Section 4.4.2.3 on page 133 formalizes the satisfaction of a contract over a path condition. That is, whether the elements in the contract are present in the path condition.

Table 4.2 repeats the three possibilities for when a contract is matched over a path condition. As well, we introduce new symbols to be employed in the truth tables in this section, which have been selected for visual clarity.

Result	**Symbol**	**Description**
Not Applicable	$\emptyset$	The path condition is *not applicable* to the contract when the pre-condition of the contract does not match.
Does Not Satisfy	$\times$	The path condition *does not satisfy* the contract when the pre-condition matches, but the post-condition does not.
Satisfies	✓	The path condition *satisfies* the contract when the pre-condition and the post-condition of the contract both match over the path condition.

Table 4.2 – Possibilities for contract proof on a path condition.

Atomic Contract. In the context of our propositional logic, contracts are referred to as *Atomic Contracts* when they contain no propositional constructs. For example, all contracts for the *Families-to-Persons* transformation (Section 7.4) are *Atomic Contracts*, such as the *AssocCity* contract in Figure 4.21.

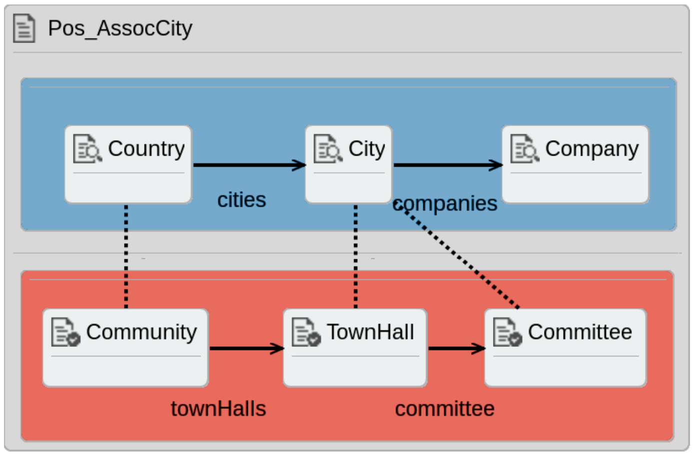


Figure 4.21 – *AssocCity* contract for the *Families-to-Persons* transformation.

In the case studies in Chapter 7, contracts are presented along with the path conditions which satisfy and do not satisfy them.

Logic Constructs. Table 4.3 details the constructs available in our contract language, along with a brief informal description.

Construct	Description
And	Both sides must satisfy for the construct to satisfy
Or	Either side must satisfy for the construct to satisfy
If-Then	If the first side is not satisfied, then the construct does not apply
Not	The results of contract satisfaction are reversed

Table 4.3 – The propositional logic operators available in the contract language.

Our propositional logic constructs are created in a recursive fashion. For example, the left-hand side of an *And* construct may be any other construct including another *And* construct. The terminal construct of this recursion is the *Atomic Contract*. Based on how each terminal *Atomic Contract* is satisfied by the path condition in question, we then build up our propositional logic with the following truth tables.

And Construct. The *And* construct is satisfied by a path condition if both sides of the construct are satisfied by that path condition. Note that this construct is commutative with respect to the two contracts on either side, and the sides are therefore denoted *A* and *B* instead of *LHS* and *RHS*.

Side A	**Side B**	**Result**
✓	✓	✓
✓	×	×
✓	∅	∅
×	×	×
×	∅	×
∅	∅	∅

Table 4.4 – Truth table for the *And* construct.

An example for this construct can be seen in the M6 contract (Figure 4.22) for the GM-to-AUTOSAR transformation. This contract is of the form *IF M6_if, THEN M6_then1 AND NOT M6_then2*, where the *If-Then* and *Not* constructs are used to join three *Atomic Contracts*.

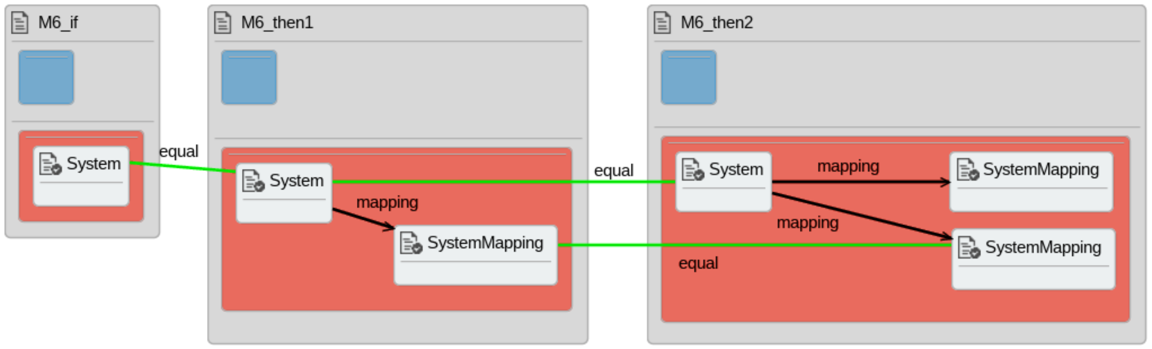


Figure 4.22 – *M6* contract.

In terms of the *And* construct, the right-hand side of the M6 contract will only be satisfied when the *M6_then1* contract is satisfied by the path condition, and the *M6_then2* contract is *Not* satisfied by the path condition. If either *M6_then1* or *M6_then2* is not applicable, then the right-hand side cannot be satisfied.

Or Construct. The *Or* construct is satisfied by a path condition if either side of the construct is satisfied by that path condition. Like the *And* construct, the *Or* construct is also commutative.

Side A	**Side B**	**Result**
✓	✓	✓
✓	×	✓
✓	∅	✓
×	×	×
×	∅	∅
∅	∅	∅

Table 4.5 – Truth table for the *Or* construct.

The *MM10* contract, described in the thesis of Selim [135], contains an example of the *Or* construct.

If-Then Construct. The *If-Then* construct is satisfied by a path condition if both the left-hand (LHS) and right-hand (RHS) sides are satisfied by that path condition. If the LHS is satisfied, but the RHS is not, then the *If-Then* construct is not satisfied. Otherwise, the construct is not applicable.

Note that a dash – for the RHS means that the result does not matter.

LHS	**RHS**	**Result**
✓	✓	✓
✓	×	×
✓	∅	×
×	-	∅
∅	-	∅

Table 4.6 – Truth table for the *If-Then* construct.

This *If-Then* construct does not correlate to a logical implication construct, as a failure to match the LHS means that the *If-Then* construct does not apply.

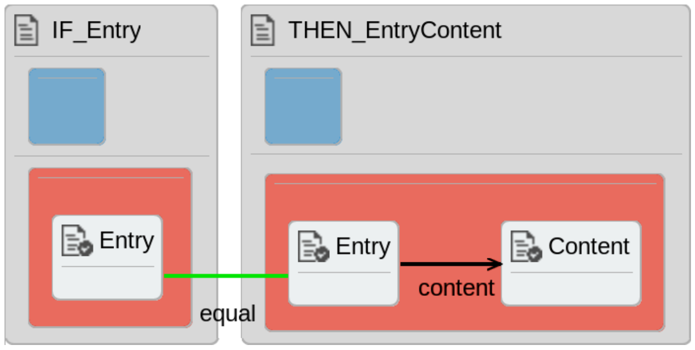


Figure 4.23 – *Entry Content* contract for the *RSS2ATOM* transformation.

There are multiple examples of this construct in Chapter 7. For example, the *EntryContent* contract for the *RSS2ATOM* transformation in Figure 4.23 demonstrates how this construct can be used to check statements of the form *if an element exists, then that element is connected this way.*

Note that to ensure that the element mentioned is the same in both the left-hand and right-hand side of the *If-Then* construct, it is necessary to use *pivots*.

Pivots. The *EntryContent* contract for the *RSS2ATOM* transformation also demonstrates the use of pivots within contract constructs. These pivots are modelled as an association between contracts, marked with the text *equals*[1].

These pivots represent elements that must be the same in both contracts for the construct to be satisfied. For example in Figure 4.23, the *Entry* element must be successfully matched by both the left-hand and right-hand side of the *If-Then* construct, and that matching must be on the same element in the path condition.

This resolution of pivot values is possible because the matching procedure returns a set of possible matches. Within our contract prover implementation, we then have

1. Note that in the thesis of Selim [135], these pivots were modelled as attributes in the elements. However, inconsistencies in naming the pivots led to faulty contracts. These errors are avoided in our current implementation approach as pivot values are automatically generated in the elements connected to the *equals* association.

a resolver which determines if the pivots are compatible for the *And* and *If-Then* constructs, and merges the sets of pivots for the *Or* construct.

Not Construct. The *Not* construct reverses the satisfiability of the contract against the path condition.

Value	**Result**
✓	×
×	✓
$\emptyset$	$\emptyset$

Table 4.7 – Truth table for the *Not* construct.

As explained in Section 4.6.3.2, contracts may employ the *Not* construct to check for the multiplicity of elements. For example, the *M6* contract in Figure 4.22 has the informal statement 'If a *System* element exists in the output model, then that *System* element should be connected to a *SystemMapping* element, and not (always) connected to two *SystemMapping* elements'. Note the *always* qualifier in that statement, which can lead to confusion.

Recall the discussion in Section 4.5.1.1 about forbidding negative elements from appearing in a contract. The reason is that the abstraction relation (Section 4.1) allows a path condition to abstract over a transformation execution which contains more input elements and rule executions. Therefore, it is impossible to say which elements will *not* be present in the input-output models abstracted by a path condition, only which elements *will* be present.

This representation may lead to misunderstanding of what a contract represents when the *Not* construct is used. When a contract is satisfied by a path condition, it is because the contract's elements are present in the path condition. The negation of this statement is that the elements are not present in the path condition as it is represented. This is not the same as guaranteeing that the elements are not present in any abstracted transformation executions.

For example, the third part of the *M6* contract checks for the presence of two *SystemMapping* elements connected to a *System* element. If these elements are found, then the *Not* construct will not be satisfied. If the elements are not present, then the *Not* construct will be satisfied. However, as the path condition only represents how many *SystemMapping* elements *must* be present in that path condition, the *Not* construct can only fail if two *SystemMapping* elements *must* be present.

Therefore, the full contract checks whether the *SystemMapping* element is connected to at least one *SystemMapping* element, and not (always) connected to two *SystemMapping* elements.

4.6.3 Types of Contracts

The thesis of Selim [135] partitions the intention of contracts into four types: *rule reachability contracts*, *multiplicity invariants*, *syntactic invariants*, and *pattern contracts*. This section will briefly discuss each type.

4.6.3.1 Rule Reachability

These rule reachability contracts determine whether a rule has been symbolically applied. The application of each rule is crucial to proper creation of the set of path conditions, as the contract or the application of other rules may depend on its application. The absence of a rule's application indicates either an error in the transformation or the requirement that a rule should be symbolically applied more than once. This is discussed in Section 4.2.3.2 and Section 4.5.

However, during development of our contract prover, it was found to be more effective to have this check performed during path condition construction. At the end of every layer, the set of path conditions is examined to ensure that every rule in the transformation from the previous and current layers is present in at least one path condition. If not, an error message is raised.

Thus, this type of contract is now automatically checked during path condition construction as described in Section 6.4.2 and no contract language construct is provided.

4.6.3.2 Multiplicity Invariants

Contracts can be used to express *multiplicity invariants* for either the source or target meta-model. This is done using the propositional logic and pivots from the contract language.

For example, as discussed above, the *M6* contract for the GM-to-AUTOSAR transformation in Figure 4.22 on page 147 has the informal statement 'If a *System* element exists in the output model, then that *System* element should be connected to a *SystemMapping* element, and not (always) connected to two *SystemMapping* elements'.

This contract is built using an *If-Then* construct, an *And* construct, and a *Not* construct. However, as previously described, the abstraction of our path condition construction means that the contract cannot guarantee the strict absence of those elements, only that they will not always be created.

Our path condition construction approach also has consequences in terms of multiplicity in that rules are not symbolically applied an infinite number of times. As discussed throughout Section 5.3.4, the number of times a rule will symbolically applied must be set at the beginning of the generation process. If the rule is not set to apply a sufficient number of times, then multiplicity contracts may fail to be applicable or counter-examples may fail to be detected.

Note that our implementation of the contract prover now includes a mechanism for determining if rule application needs to be repeated, as discussed in Section 6.4.1.

4.6.3.3 Syntactic Invariants

Syntactic invariant contracts check whether the path condition is well-formed with respect to the input or output syntax. These contracts therefore have either an empty pre-condition or an empty post-condition.

An example of this type of contract is the SS1 contract in Figure 4.24 for the UML-to-Kiltera transformation. The informal statement for this contract is 'if there is an *Inst* element, then that *Inst* element has a similar name to a *ProcDef* element.' The defining feature making this contract a syntactic invariant is that this contract

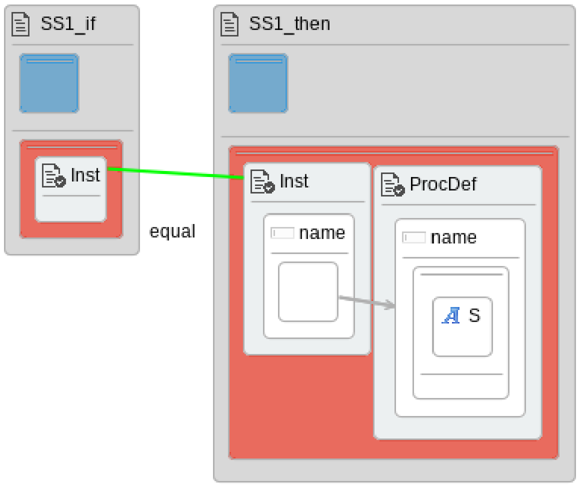


Figure 4.24 – *SS1* contract.

only constrains elements in the output model of the path condition, without making reference to the input model.

4.6.3.4 Pattern Contracts

The final category of contracts described by Selim are *pattern contracts*. In these contracts, the elements in the input model are related to elements in the output model. Put another way, these contracts are of the form, 'If these elements exist in the input model, then these elements must exist in the output model.'

Thus, the intention of these contracts is to verify that the rules in the transformation are interacting in a valid and predictable way to produce the correct output elements. This verification is difficult to perform from manual inspection of the rules themselves.

All the contracts presented in Section 7.4.1 for the *Families-to-Persons* transformation are examples of pattern contracts.

4.6.4 Limitations

Section 7 presents contracts for each of our case studies, which demonstrates their use in understanding and detecting errors in a transformation. However, there are a number of significant limitations to our contract language, mainly due to its organic creation during our case studies.

First, our contract language is currently limited to represent only structural conditions and not arbitrary expressions. Thus, it is much less powerful than other constraint languages such as OCL [170]. In particular, it is impossible to represent such constructs as sets operations, non-string attributes, or helper functions in our contract language. The lack of a *for-all* operator means that our contract language may not even be as powerful as first-order logic [71],

The nature of the symbolic execution approach also restricts the contracts which can be written. As discussed in Section 4.6.3.2, the contract prover symbolically executes rules a particular number of times. A contract may be incorrectly satisfied or not satisfied based on this limit, and the user must be extremely careful about the use of multiplicity invariants and the *Not* operator.

Another restriction on our contract language imposed by the symbolic execution approach is the lack of validation of input or output attributes. As the contract prover operates on the transformation specification, without a defined input model, path conditions can only refer to abstract input attributes. Therefore, it is impossible to write a contract to validate that (as an example for the *Families-to-Persons* transformation) all *Family* elements have a *lastName* that starts with a capital letter.

The notion of time or order is also difficult to express in our contract language, if not impossible. For example, we attempted to design a contract for the *mbeddr* transformation which would determine in which order elements were connected in the output model. However, as the semantics of DSLTrans rules means that all output elements are created at essentially the same time (Section 3.4.1), there is no order to how connections were made.

One way of approaching the notion of time or order is to have contracts that are checked at the end of each layer during path condition construction, rather than once the entire transformation has been symbolically executed. However, it is unknown how this would meaningfully increase the expressibility of the contract language.

We note that it may be useful to determine how intermediate structures are built, but this is less useful when element deletion is not represented. As well, there is a performance cost, as intermediate contracts would have to be proved against every path condition.

Another avenue to explore in future work is the integration of constructs from other property languages. In particular, the property language of the *ProMoBox* approach [103] employs temporal patterns which express constraints such as *eventually* or *until*. It is a future research question whether these types of properties are relevant to the *translation* transformations which our approach verifies, and how our approach can verify them.

While our contract language allows for the definition of a wide variety of structural conditions, the limitations discussed here do restrict the expressibility. Future research into contract expressibility will attempt to align (or replace) our approach with that of Guerra *et al.* [69], where the authors use PaMoMo as the basis for their contract language. This specification allows the authors to define a variety of visual contracts to express both negative and positive contracts in an elegant approach.

4.7 Conclusion

This chapter has presented our technique for proving pre-condition/ post-condition contracts on a DSLTrans transformation. This technique relies on the notion of symbolic execution, which creates path conditions which represent valid combinations of rules during a transformation execution. This representation is through the abstraction relation (Section 4.1), which defines how the elements in a path condition imply the existence of elements in an input-output model pair.

The path condition generation process is discussed in Section 4.2. This process moves through the layers in a transformation to build the path conditions by resolving the dependencies present between path conditions and rules, and combining them in different ways. In this thesis, the path condition generation process is placed in the double-pushout approach. This approach allows the creation of mass productions for rules and layers, which simplifies the explanation considerably. As well, this approach allows the integration of additional constructs, such as indirect links, negative elements, and the representation of multiple applications of each rule in the path condition. These constructs are critical to define as they have consequences for contract proving.

This chapter also presented our technique for matching contracts on path conditions (Section 4.4), which is done through the typed graph split morphism. This morphism allows for a node in the contract to be 'split' over multiple nodes in the path condition. The use of this morphism allows our technique to forego the use of explicit representation of all nodes merging, which was seen in past work to be computationally expensive.

The abstraction relation allows the results of contract proof on a path condition to be correctly extended to the input-output models represented (Section 4.5). Our technique can prove contracts on a finite set of path conditions, and obtain the set of path conditions where the contract is satisfied, and path conditions where it is not satisfied. These path conditions can then be examined to better understand the transformation.

Finally, this chapter discussed the constructs available in the contract language in Section 4.6. Examples are provided from the case studies we examine in this thesis to illustrate the use of each construct.

Chapter 5
SyVOLT Design and Work-flow

This chapter discusses the SyVOLT tool, which is our implementation of the symbolic execution process mentioned in Section 4.2 and the contract proving approach discussed in Section 4.4.

The guiding principle of the Modelling, Design, and Simulation Lab of McGill University (`msdl.cs.mcgill.ca`), where our transformation verification research originated, is that all tools and processes are explicitly modelled at multiple levels of abstraction with multiple formalisms. This complexity management is the driving force of *multi-paradigm modelling* (MPM) [104, 60], which is an approach to leverage numerous modelling languages and tools to decompose complex problems. This principle ensures that essential complexity is the main object of software development and that accidental complexity is mitigated as much as possible.

This chapter explains the work-flow of the SyVOLT tool, highlighting where the principles of multi-paradigm modelling have been applied. For example, Section 5.1 provides an overview of our tool in graphical form as a *Formalism Transformation Graph and Process Model* (FTG+PM). This diagram allows the succinct representation of the work-flow of the tool, as well as the myriad of formalisms that are employed.

The FTG+PM also relates how many of the model manipulations within the SyVOLT tool are encoded as model transformations. In particular, the underlying representations of the artefacts (DSLTrans transformations, the SyVOLT contracts, and the path conditions produced) are all modelled explicitly and manipulated by the T-Core framework [145] (Section 5.3.2). Initial research presented by Lúcio and Vangheluwe [99] explains how we use model transformations to verify model transformations.

We present these details of the prover for a variety of reasons. First, this provides groundwork for the discussion of algorithmic optimizations in Chapter 6, which are a contribution of this thesis. Second, we aim to promote the use of model-driven development within modelling tools, which increases modularity, and with proper formalization our algorithms and optimizations can be transferred to other model-manipulating tools.

As an example of the benefit of this approach, the modularity of the SyVOLT prover allows us to easily create new front-ends to build transformations and contracts for verification (Section 5.2). As an example, we have created a higher-order transformation to generate transformations automatically from ATL (Section 5.2.1), as well as graphical editors in the Eclipse (Section 5.2.2) and MPS environments(Section 5.2.3).

These front-ends allow for the user to easily create DSLTrans transformations and contracts for verification. The SyVOLT tool implementation (Section 5.3) is then started from within the front-end. The results produced are then returned to the user, allowing quick iteration of transformations and contracts.

Figure 5.1 on the following page shows a high-level overview of the architecture of SyVOLT. The grey boxes represent the top frontends and the backend.

Thus, this chapter answers our fourth research question: *What is the design and work-flow of a contract verification tool?*

5.1 FTG+PM

The SyVOLT contract prover operates on many different artefacts (matchers, rewriters, meta-models) in a variety of formalisms (Ecore, Himesis) as each

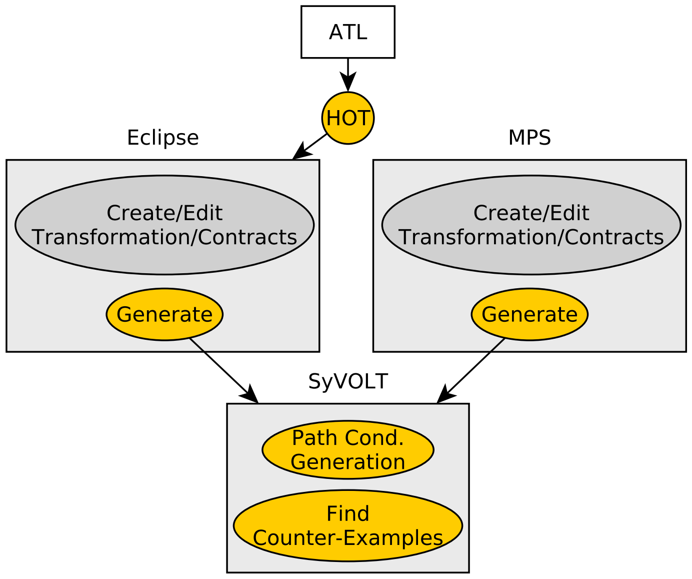


Figure 5.1 – Overview of the front-ends and back-end of the SyVOLT architecture.

representation has a specific focus. To represent this diversity and establish a user's work-flow to prove contracts, we present in Figure 5.2 a graphical depiction of SyVOLT's operation.

The *Formalism Transformation Graph and Process Model* (FTG+PM) [95] has the aim of representing the entirety of the model-driven engineering process by a) representing the domain-specific formalisms and the transformations between them, and b) representing the process and data flow from the start to the end of the modelling activity. The intention is that by explicitly modelling and uniting these flows, users can better understand the system under study. As well, the diagram itself could be treated as a model for the purpose of automation. A case study for the automotive domain is presented in [106].

The top portion of Figure 5.2 is the *Formalism Transformation Graph* (FTG). This graph represents the formalisms present in the process as white labelled rectangles, while the transformations between them are represented as labelled ovals. Note that here the term transformations is used rather loosely, as it may include not

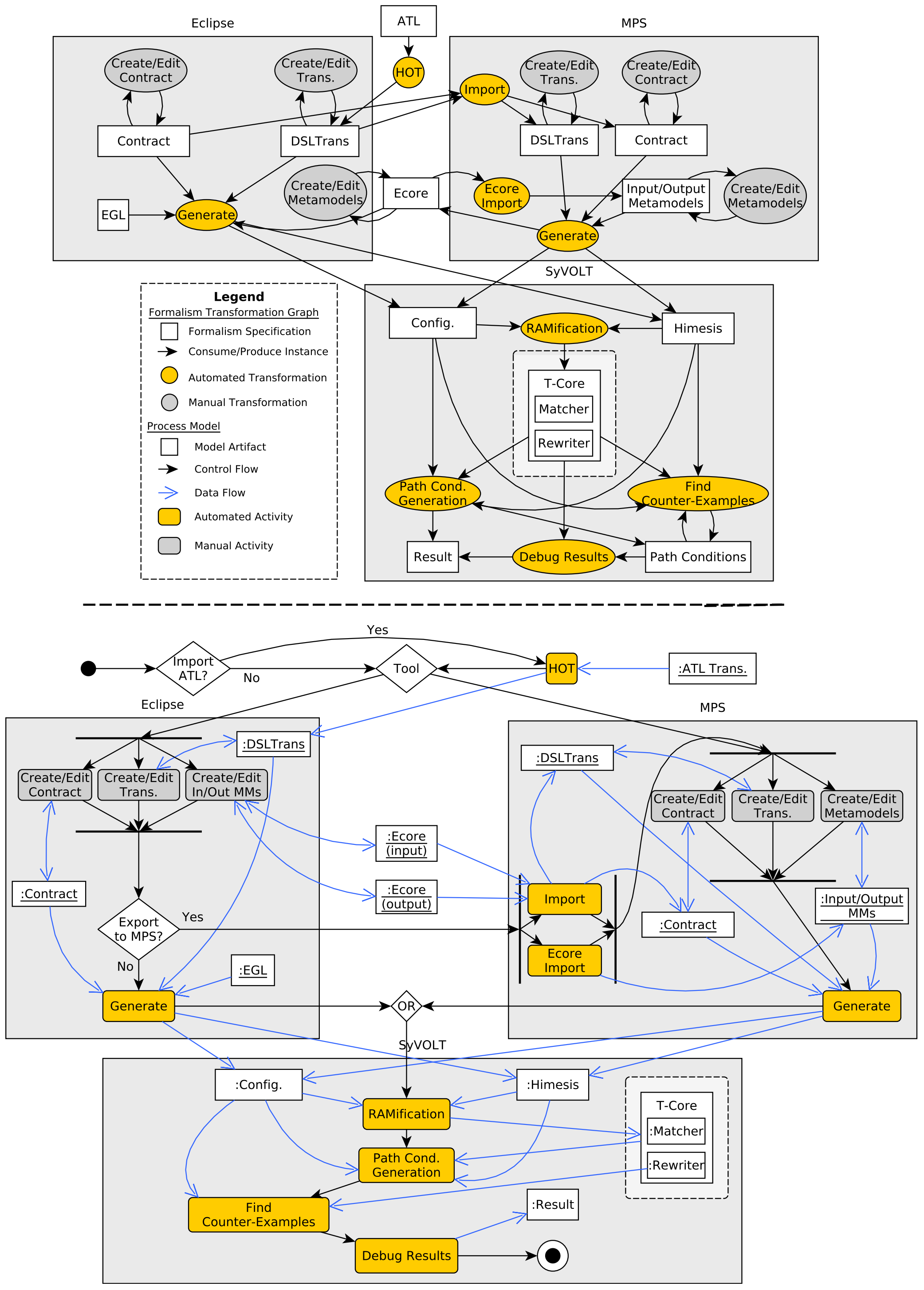


Figure 5.2 – The FTG (top) and PM (bottom) of the contract proving process.

only a model-to-model transformation, but also model-to-text transformations, a manual editing activity, or other model changes. The transformations are divided into two colours: the grey ovals represent manual activities that need human intervention, while the yellow ovals are automated transformations. The arrows between the formalisms and the transformations represent the formalisms that the transformations takes as input instances and produces as output instances.

The Process Model (PM) on the bottom of Figure 5.2 precisely specifies both the process and data flow of the contract proving activity using the UML Activity Diagram 2.0 language [115]. For the process flow, the rounded rectangles represent executions of the transformations with the same label in the FTG side of the figure. As in the FTG, grey symbols represent transformations requiring human intervention, while yellow symbols represent automatic transformation execution.

Control flows along the thick black lines, where it may fork and join at thin black bars. As well, diamonds represent decision points in the activity. Along with the control flow, the data flow is also represented by the PM. Thin blue lines represent the instances of the formalisms which are produced and consumed by the transformations in the activity.

The SyVOLT contract prover currently has three components: an Eclipse front-end, a MPS front-end. and a Python back-end. To represent these three components, we have separated both the FTG and the PM in Figure 5.2 into three marked sections. A further explanation of the technology, reason for adoption, and images for both front-ends is available in Section 5.2.2 and Section 5.2.3.

Transformation and Contract Construction. We begin our explanation of the FTG+PM in Figure 5.2 with the PM side on the bottom. First, the user must decide if they wish to convert an ATL transformation [3] into the DSLTrans language using a higher-order transformation (Section 5.2.1). This automatic transformation takes as input a transformation written in the ATL language, and produces a DSLTrans transformation.

The second decision for the user is whether they wish to edit the transformation and contracts within the Eclipse environment or within the MPS environment. Note that the user can also decide to export the transformation and contracts from Eclipse to MPS at a later time, but that the reverse direction is not implemented at this time.

The user then works within the Eclipse or MPS environments to create or edit the transformation and contract artefacts as they wish. In Eclipse, users are also able to directly edit the input and output meta-models files, which are stored as Ecore files [141]. On the MPS side, a transformation is available to import Ecore files as a language within the MPS editor so that this language can be edited as desired. The advantage of this approach is that the transformation and contracts are then strongly-typed by the meta-models, which has been instrumental in detecting a number of errors with our case studies. This is further discussed in Section 5.2.3.

Following the editing activities, the user can then begin the contract proving process itself. On the Eclipse side, the transformation and contracts undergo a transformation using the Eclipse Generation Language (EGL) [6, 130] which produces the required artefacts for the Python back-end. In MPS, the model-to-text language is integrated directly into the specifications of the transformation and contract languages.

SyVOLT Tool Implementation. The SyVOLT tool is represented by the grey container at the bottom of both the FTG and PM in Figure 5.2. Note that all artefacts and formalisms in the back-end are implemented in Python code.

The generation processes from the front ends will produce the two types of artefacts required for the SyVOLT back-end. The first type is a configuration file, which contains details about the proving process such as the transformation's layers and the contracts to be proved. The second type is a set of attributed typed graphs to represent the rules in the transformation and the contracts. These typed graphs are stored in the Himesis format (Section 5.3.1), which efficiently encodes the input, output, and linking structures that we use throughout the proving process.

Our implementation requires the use of T-Core *matchers* and *rewriters* (Section 5.3.2), which are transformation language primitives used to detect when rules can combine with other rules and to create the typed graphs representing the path conditions themselves. The PyRamify transformation (Section 5.3.3) creates these matchers and rewriters directly from the Himesis representation of the transformation rules.

Next, the path condition construction process (described in detail in Section 5.3.4) will operate on the Himesis typed graphs, the configuration file, and the T-Core matchers and rewriters to generate the set of path conditions for the transformation. Our technique can then find counter-examples by applying the matchers for the contracts and producing those path conditions which succeed or fail for each contract. Finally the contract debugger transformation produces debugging output for the user to understand the counter-example results. At this point, the user may then modify the transformation or contracts as needed to iterate on this process.

5.2 Transformation and Contract Construction

The starting point for operating our transformation verification tool is the construction of the DSLTrans transformation and the contracts to be verified on that transformation. As shown in Figure 5.2. there are numerous ways to accomplish this. In this section, we will describe a) our approach to converting transformations from the Atlas Transformation Language (ATL) [74] to DSLTrans using a higher-order transformation, b) our plug-in developed for the Eclipse ecosystem [5], and c) the development of the DSLTrans and contract language within the Meta-Programming System (MPS) [8]. Included in each section is a discussion of the technologies involved and the reason for developing that approach.

The Eclipse and MPS plug-ins are used to interface with the back-end of the SyVOLT tool, which is discussed in Section 5.3. Generation scripts in both plug-ins are available to automatically produce the artefacts required and begin the contract proving process.

5.2.1 Higher-order Transformation from ATL

A core benefit to employing a model-driven engineering process is the ability to develop *higher-order transformations* which take as input one modelling language and produce another [28, 164]. The usability of our tool has been extended by defining one of these higher-order transformations to one of the most prominent model transformation languages in the model-driven engineering community, which is the Atlas Transformation Language (ATL) [3, 74]. In fact, ATL has become a de-facto standard for implementing model transformations. The success of ATL is due to many factors: the flexibility and expressiveness of the language, support for the main meta-modelling standards, high usability, excellent tool integration with the Eclipse environment, and active research and development communities.

It is understood by the ATL community that improving the verification methods available would allow more widespread adoption into industrial and academic practices. To this end, we have previously developed a technique to convert declarative ATL model transformations[1] into an equivalent DSLTrans version [113]. This conversion process is possible through the definition of a higher-order transformation which is also written in ATL. The produced transformations are then available for further editing or proving activities in either the Eclipse or MPS front-ends.

This section will briefly describe ATL by presenting selected rules from the ATL version of the *Families-to-Persons* transformation, which is discussed in Section 7.4. Following this, we will briefly discuss the higher-order transformation itself and the research questions which have been answered by our previous works.

5.2.1.1 Atlas Transformation Language

The ATL transformation language is similar to other transformation languages, in that a transformation is composed of rules which match elements in an input

1. Declarative transformations lack the imperative constructs (assignments, loops, if statements) discussed in the ATL User's Guide: `https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#ATL_imperative_code`

model and produce or rewrite elements in an output model. However, ATL is far more expressive than the DSLTrans language.

For example, the DSLTrans language is strictly an out-place transformation language, meaning that the transformation language cannot change the input model. ATL does not have this restriction, making it possible to write simulation transformations. Another difference is that ATL is neither layer-based or guaranteed to terminate like DSLTrans (Section 3.4). Finally, ATL includes the Object Constraint Language (OCL) [170] to define set operations and complicated traversals when matching and setting attribute values.

Listing 5.1 on the next page shows three selected rules from the ATL version of the *Families-to-Persons* transformation. As ATL operates on typed graphs, the input and output meta-models *Families* and *Persons* are specified at the top of the transformation.

The matching portion of an ATL rule is located under the *from* text, and is called the *in-pattern*. It contains *in-pattern elements* along with an optional filter on attributes. These in-pattern elements will match over the elements in the input model as long as they satisfy the filter. The rewriting portion of the rule is under the *to* text, and this *out-pattern* contains the *out-pattern* elements which will be created or modified in the output model. These out-pattern elements have *bindings*, which are used to initialize the attributes or references of the the out-pattern elements.

For example, the *Country2Community* rule in Listing 5.1 matches over a *Country* element from the *Families* meta-model, and produces a *Community* element from the *Persons* meta-model. On the output model, there are a number of references created. The *townHalls* references are set from the later execution of a rule which creates *TownHall* instances from *City* elements. The set of *persons* references are created by performing the union of four *collect* operations, which collect all *fathers*, *mothers*, *daughters*, or *sons* who belong to *Families* in the matched *Country*. Finally, the *associations* references are set from the *Association* produced from those *Companies* which are in *Cities* belonging to the matched *Country*.

Listing 5.1 – *Families-to-Persons* ATL Transformation

```
module Families2Persons;
create OUT:Persons from IN:Families;

rule Country2Community {                           --R1
 from
  c: Families!Country
 to
  cmm : Persons!Community (
   townHalls <- c.cities,
   persons <- c.families->collect(f|f.fathers),
   persons <- c.families->collect(f|f.mothers),
   persons <- c.families->collect(f|f.sons),
   persons <- c.families->collect(f|f.daughters),
   associations <- c.cities->collect(cty|cty.companies
    -> collect(cmp|Tuple{ct=cty,cm=cmp}))    -- navigation
  )}

rule Father2Man {                                  --R2
 from
  p : Families!Parent
   (p.family.fathers.includes(p))            -- filter
 to
  m : Persons!Man (
   fullName <- p.firstName + " " + p.family.lastName -- attribute setting
  )}
```

Notable in the *Father2Man* rule is the attribute setting of the *fullName* on the last line of Listing 5.1. The new value of the attribute is set as the concatenation of the *firstName* of the person and the *lastName* of the family of the person.

Another important construct in ATL (which is not shown here) is a call to a *lazy rule*, which can perform complicated computations similar to a function call in a programming language.

5.2.1.2 Higher-order Transformation

As ATL transformations conform to the ATL language, it is possible to manipulate them by means of a higher-order transformation (HOT) [28]. Figure 5.3 shows the meta-models involved in this operation. Dashed arrows represent a 'conforms to' relationship, while solid arrows represent the input and output of the transformation. On the left is the ATL transformation which conforms to the ATL meta-model which contains the constructs in the ATL language. The higher-order transformation, which also conforms to the ATL meta-model, is able to parse the ATL transformation and create an equivalent DSLTrans transformation. This DSLTrans transformation conforms to the DSLTrans language, which contains concepts such as layers, rules, *MatchModels*, and so on.

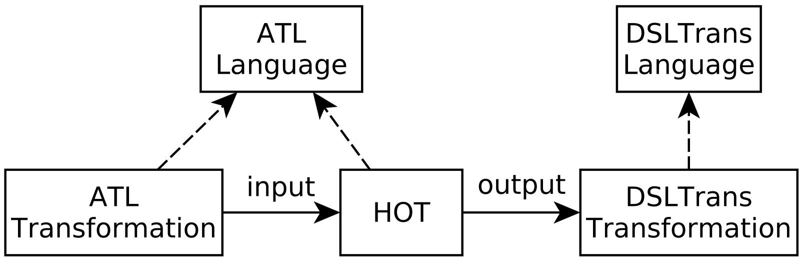
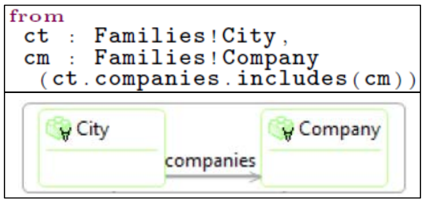


Figure 5.3 – Conformance and data relationships between a ATL model transformation and the higher-order transformation which produces a DSLTrans version of that transformation.

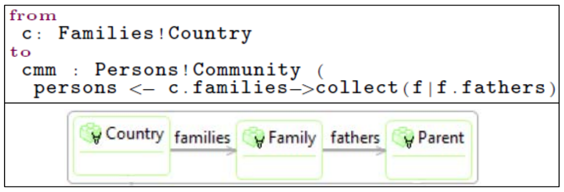
The higher-order transformation matches elements in the ATL transformation such as filters, OCL expressions, and lazy rules, and produces the equivalent versions in the DSLTrans specification. Examples of these conversions are seen in Figure 5.4. In Figure 5.4a, the ATL code is selecting those *City* and *Company* elements where the *Company* is found in the *City*. Figure 5.4b shows how OCL constraints can be used to select elements. In that example, the *fathers* of a *Family* element are selected.

Further details on these example are found in our previous work [113], including a detailed description of the conversion of the *Families-to-Persons* transformation from ATL to DSLTrans.



```
from
 ct  :  Families!City,
 cm  :  Families!Company
  (ct.companies.includes(cm))
```

(a) *CityCompany2Association* match graph created from filter elements.



```
from
 c: Families!Country
to
 cmm  :  Persons!Community (
  persons <– c.families–>collect(f|f.fathers)
```

(b) *copersons[..]* rule created from OCL bindings.

Figure 5.4 – Examples of converting ATL rules to DSLTrans match elements.

5.2.1.3 Transformations Produced

For our research on converting ATL transformations to DSLTrans, we were guided by two important research questions. The first was, *Is our higher-order transformation technique applicable to complex ATL transformations?*, while the second was, *does the version of the transformation produced by our higher-order transformation differ significantly from a hand-built transformation?*.

For the first research question, we answered in the affirmative as we have conversion results for a variety of transformation and meta-model sizes [113]. In fact, the higher-order transformation is able to convert many ATL transformations, given a short list of restrictions: transformations must be terminating, confluent, declarative. out-place, not use unique lazy rules, and not have complex attribute setting.

Note that these restrictions are not significant for the domain of translation transformations. For example, using the declarative part of ATL normally results in clearer, more readable and more maintainable model transformations than when the imperative part of the language is used. As well, the out-place use of ATL is significantly more common [151] than the in-place *refining mode* [150].

We answered the second research question in the negative, based upon our case studies. With the permission of our collaborators at Queen's University, we were able to convert the ATL representations of the *GM-to-AUTOSAR* and *UML-to-Kiltera* transformations to DSLTrans. Note that these are the transformation versions presented in Section 7.2 and Section 7.3. These automatically-converted transformations were compared to the hand-built versions presented in the previous work of [137] and [135]. The produced transformations were very similar, and produced identical verification results for the contracts. Thus, we declared that the higher-order transformation can produce DSLTrans transformations of equivalent quality to hand-built transformations.

In the future, we intend to integrate the higher-order transformation more closely within the DSLTrans front-ends in Eclipse or MPS, such that it is automatically invoked whenever the user wishes to load an ATL transformation. This would increase usability and adoption of our tools.

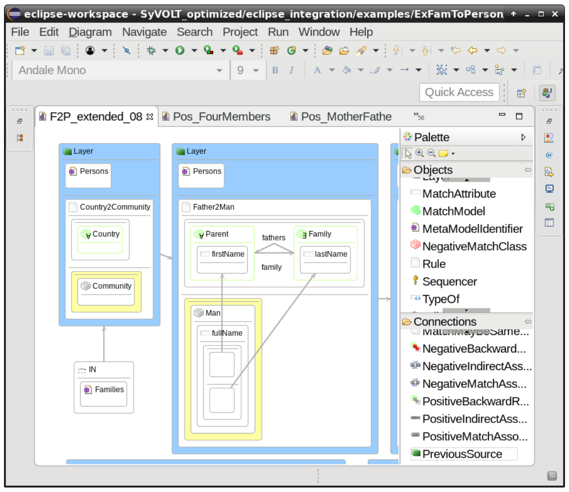
A second direction of research is to improve the higher-order transformation itself. Currently, the higher-order transformation is quite complicated and consists of 2257 lines of ATL code. This complexity makes it extremely difficult for a user not versed in ATL to maintain, and precludes the higher-order transformation from being transformed itself into a DSLTrans transformation.

5.2.2 Eclipse Plugin

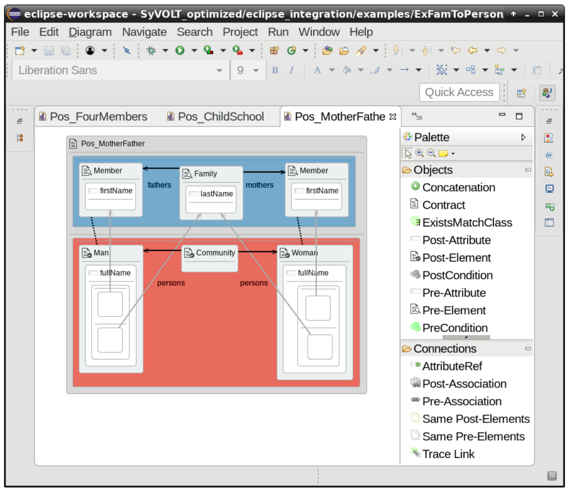
This section describes the plugin for the Eclipse environment which allows the user to build transformations and contracts using a graphical editor, and then begin the contract proving process.

The Eclipse platform offers a rich ecosystem with opportunity for customization and a vibrant developer community [5]. As well, there are multiple model transformation tools offered as plugins, including ATL [3], a DSLTrans transformation engine [21], the Epsilon modelling framework [6], VIATRA [25] and EMF-IncQuery for performing model queries [153]. These tools benefit from being integrated with the Eclipse Modelling Framework (EMF) [141], which is a structured meta-model for the Eclipse environment such that multiple tools and work-flows can build on a common modelling and code generation framework and inter-operate more freely.

In collaboration with Cláudio Gomes from the University of Antwerp, we have developed a graphical interface to build transformations and contracts directly within the Eclipse editor, as seen in Figure 5.5. On the right-side side of the editor is the palette, which contains the objects and connections used within the artefacts. These elements can be created and edited freely on the canvas.



(a) *Families-to-Persons* transformation.



(b) *MotherFather* contract.

Figure 5.5 – Editor for DSLTrans transformations and contracts within Eclipse.

5.2.2.1 Technology

The technology behind the plugin is the Graphical Modelling Framework (GMF) [7]. This framework allowed us to quickly create visual editors for the transformation and contract language, based on model representations of those languages built using the EMF. With this plugin, transformations and contracts can be quickly created.

However, a criticism of this effort is that the editors lack polish, and no consistency checks are made between the artefacts and the input and output meta-models. This lack of consistency led to a number of errors in our case studies, such as the name of an element being slightly different in the contract versus the transformation. In this case, the contract could not be satisfied by any path condition. In contrast, our plugin for the MPS platform (Section 5.2.3) has these consistency checks, which we note to improve the creation time and decrease errors for transformations and contracts.

5.2.2.2 Artefact Generation

Once the transformation and the contracts have been created by the user the generation of artefacts for the contract prover back-end (Section 5.3) can begin. This process is orchestrated by the Ant build tool [1], which can orchestrate the multiple loading and generation steps required. The Epsilon Generation language [130], which is a template-based model-to-text language, creates the artefacts required for the SyVOLT prover. The Python script then begins prover operation. The prover's output appears in the Eclipse output pane, showing the user the current status, counter-examples as sets of rule names, and the contract result analysis information (Section 6.4.3).

5.2.3 Meta Programming System (MPS)

As our contract proving technique is aimed towards model-driven engineering, we have also developed a plugin for a development environment focused on the language engineering and verification community. This environment is the Meta Programming System (MPS) editor [8] from Jetbrains, which is designed for the rapid creation and use of *domain-specific languages* (DSLs), which can ease the development of software [159].

The language verification plugin provides the DSLTrans transformation language constructs necessary to build transformations and rules, along with the SyVOLT language constructs for building contracts. The plugin is available from within the MPS plugin repository under the name *DSLTrans*, or at the repository site [96].

5.2.3.1 Building Domain-Specific Languages

MPS is a *language workbench* in that languages can be easily created and extended [53]. As part of our research, we re-created the DSLTrans and contract languages within MPS to provide the necessary concepts to build transformations and contracts. Each language contains powerful and built-in aspects, allowing the building of powerful domain-specific languages.

For example, a language definition in MPS contains different aspects for:

— *Structure* - The structure aspect is essentially a class definition with attributes and references.

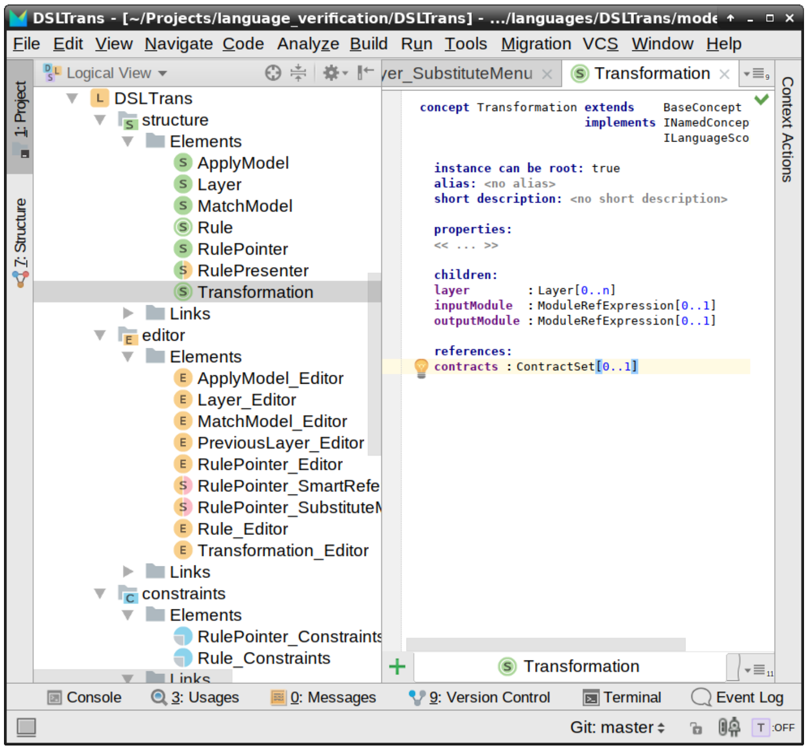
— *Editor* - The editor aspect defines the visual presentation of each element in the editor.

— *Constraints* - The constraints aspect allows for the definition of arbitrary constraints using a Java-like language on the values for element attributes/references.

— *Behaviour* - Arbitrary Java-like code for use by actions or plugins is defined in the behaviour aspect.

— *TextGen* - The text generation aspect provides built-in support for model-to-text generation.

As an example of a language definition, we present in Figure 5.6 an MPS window with the DSLTrans language definition open. In the left-hand pane are the aspects for the DSLTrans language, which allow us to define the constructs in the language, their visual appearance, and constraints. In the right-hand pane is the structure aspect of the *Transformation* concept within the DSLTrans language. Note that the transformation contains a set of layers, a pair of input and output models (for executing a transformation), and a reference to the set of contracts to prove on this transformation.



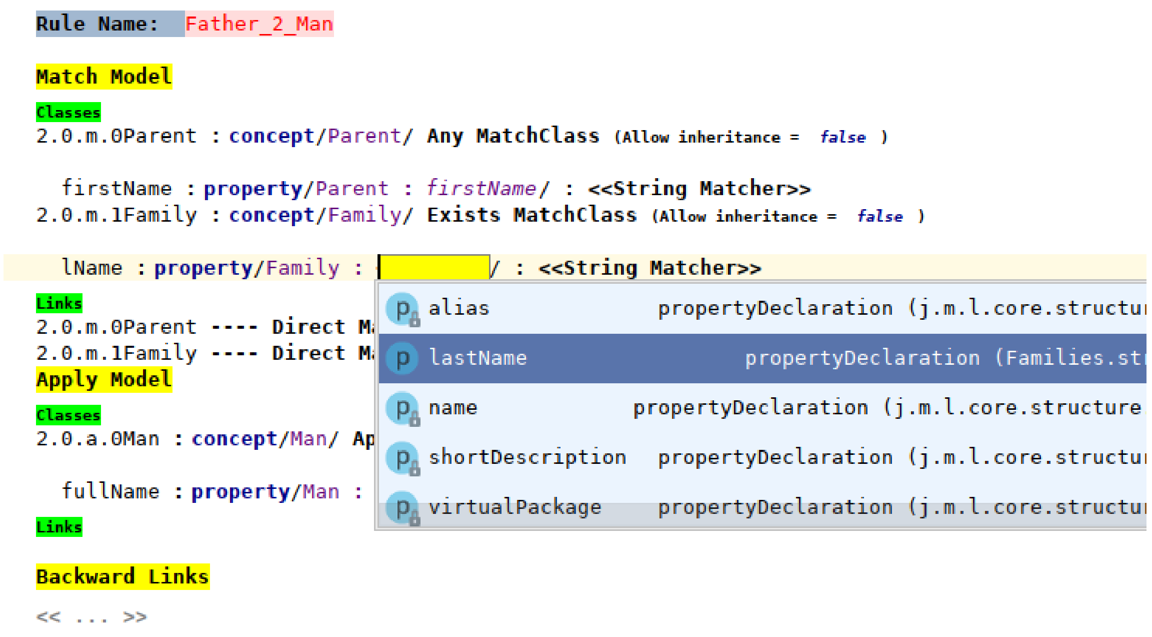
```
concept Transformation extends    BaseConcept
                       implements INamedConcep
                                  ILanguageSco

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  layer        : Layer[0..n]
  inputModule  : ModuleRefExpression[0..1]
  outputModule : ModuleRefExpression[0..1]

  references:
  contracts : ContractSet[0..1]
```

Figure 5.6 – Language definition aspects in MPS and the *Transformation* concept.

5.2.3.2 Projectional Editing

MPS is a *projectional editor* [54], which means that the user does not edit the underlying *abstract source tree* of a program, but instead a projection. The advantage of this approach is that it is possible to reason about each component of the abstract source tree at a higher level than just editing text, enabling advanced constraint checking and auto-completion. This projectional editing also combines with the power of explicit modelling of languages to provide a number of desirable features, such as auto-completion and type checking in the projectional editor.

We have developed constraints for the DSLTrans language in MPS such that elements in the match and apply models of each rule must be instances of a concept in the input or output languages, which are explicitly modelled within MPS.



```
Rule Name:  Father_2_Man

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

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

  lName : property/Family : |        / : <<String Matcher>>
Links                        alias               propertyDeclaration (j.m.l.core.structu
2.0.m.0Parent ---- Direct M
2.0.m.1Family ---- Direct M  lastName                  propertyDeclaration (Families.st
Apply Model
Classes                      name              propertyDeclaration (j.m.l.core.structure
2.0.a.0Man : concept/Man/ Ap
                             shortDescription    propertyDeclaration (j.m.l.core.structu
  fullName : property/Man :
Links                        virtualPackage      propertyDeclaration (j.m.l.core.structu

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

Figure 5.7 – Editing the *Father2Man* rule with auto-complete in MPS.

For example, Figure 5.7 demonstrates the *Father2Man* rule, which includes a *Parent* element and a *Family* element. In the figure, the user is in the process of adding an attribute to the *Family*, and the auto-complete pop-up indicates that the *Family* concept includes the *name* and *lastName* as valid attributes. Thus, all elements and attributes in the transformation and contracts are typed according to the input and output languages. We (and others [168]) note that this deep integration of languages allows for quick and accurate creation of code/transformations.

Figure 5.7 also shows an example of a constraint defined in the DSLTrans language. At the top of the figure, the rule name is highlighted in red as it violates the constraint that rule names may not contain an underscore. This constraint is present purely for technical reasons, but it showcases how arbitrary constraints can be written for elements. Each element in the language can be passed to a constraint method, where a Java-like language determines whether the element value is allowed or not.

5.2.3.3 Plugins Created

An alternative to the creation of transformations and rules by hand within MPS is to employ a DSLTrans importer [96] that we have created as part of this work. This plugin is able to parse a DSLTrans transformation or contract file and create the appropriate transformation or contract model.

The MPS framework allows the plugin to directly instantiate elements of the input and output languages such as attributes and associations to place within the model. This allows for the creation of powerful higher-order transformations to be distributed as editor plugins.

Another contribution of this research has been the development of the *EcoreImport* [109], which is able to quickly import a meta-model stored in an Ecore file into the structure file of an MPS language. This improves the inter-operability of the SyVOLT tool with other modelling frameworks.

5.2.3.4 Artefact Generation

The generation of artefacts for the prover back-end in the MPS environment is similar to that performed in the Eclipse environment (Section 5.2.2). However, as opposed to the Eclipse Generation Language scripts, the model-to-text generation step is defined by the DSLTrans language in the TextGen aspect.

After model-to-text generation has created the configuration file and Himesis artefacts for the transformation, rules, and contracts, the back-end of the prover is started. The contract proving results and analysis will then appear in MPS as a pop-up window.

5.3 Proving Process Implementation

This section will discuss multiple facets of the implementation of our contract prover. In particular, we will discuss both the high-level components as well as the low-level format that our prover manipulates.

We begin our description with the low-level Himesis format in Section 5.3.1, which is the representation for typed graphs. Following this, we explain the T-Core framework for model transformation primitives in Section 5.3.2. This framework offers the *Matcher* and *Rewriter* concepts necessary to manipulate the Himesis graphs and create path conditions.

We then explain the crucial *PyRamify* component of our contract prover in Section 5.3.3. This component is primarily responsible for taking the Himesis files which represent rules in the transformation, and creating their respective T-Core *Matcher*. As explained in that section, a *Matcher* is produced for each rule which would successfully match over that same rule. Thus, the *PyRamify* component implements a form of RAMification [85], which creates a pattern specification language for a meta-model. In this case, a language is produced which can match over rules themselves.

Finally, in Section 5.3.4 and Section 5.3.5 we provide a few details on the implementation of the path condition construction process and contract proving steps defined in Section 4.2 and Section 4. In particular, we discuss how the T-Core Matchers and Rewriters are employed to build up the set of path conditions and prove contracts.

5.3.1 Himesis Format

The Himesis framework is a graph-based kernel implemented in Python, as discussed in the Master's thesis of Provost [122]. Himesis contains concepts for representing pre- and post- conditions for graph matching, as well as efficient algorithms for isomorphic graph matching. Significant emphasis in the design and implementation of Himesis was placed on performance, especially that of attribute matching [144].

Himesis is an essential part of our contract prover implementation, as all rules, contracts, matcher graphs, rewriter graphs, and path conditions in our contract prover are represented in Himesis. Therefore, the next section will show an example of the Himesis format to ground further discussion of operation of T-Core (Section 5.3.2) and the split morphism matching algorithm (Section 6.2).

5.3.1.1 Himesis Example

The Himesis format is based on that of *igraph* format [11], which is a widely-used open-source graph library. As an example, we will show the Python code which generates the Himesis graph for the *Father2Man* rule in Listing 5.8 on the following page. The *Father2Man* rule itself is shown in Figure 5.9 on page 179 for reference.

The Lines 1-8 are for initialization, where the proper modules are imported, the class is defined, and the rule is initialized with the proper name and global ID. Note that the graph is instantiated as its own class, which is a subclass of the *Himesis* class. An interesting facet of the Himesis format is that the nodes and edges for the graph can be created at initialization time. This allows the file to be loaded dynamically by the contract prover and facilitates the use of model-to-text generation languages.

Lines 10 to 15 create the nodes for the *MatchModel* and *ApplyModel* for this rule. These nodes are container nodes for the rule elements within the match and apply components of the rule. Note that the *mm__* attribute denotes the type of the node in the rule meta-model.

As well, there is a *paired_with* node defined on Lines 16-19. This node will be attached by edges to both the *MatchModel* and *ApplyModel* nodes to indicate that they belong to the same rule. This is important as multiple rules can exist within a path condition. An example of attribute setting for a node is seen on Line 20, where the rule's name is stored in the *attr1* attribute for the *paired_with* node.

The *Parent* and *Family* elements for the match model are created in Lines 21 to 26, while the *Man* element in the apply model is created on Lines 28 and 29.

```
import uuid
from core.himesis import Himesis
class HFather2Man(Himesis):
    def __init__(self):
        super(HFather2Man, self).__init__(name='HFather2Man', num_nodes=0, edges=[])
        self["mm__"] = ['HimesisMM']
        self["name"] = """Father2Man"""
        self["GUID__"] = uuid.uuid3(uuid.NAMESPACE_DNS,'Father2Man')

        # match model node
        self.add_node()
        self.vs[0]["mm__"] = """MatchModel"""
        # apply model node
        self.add_node()
        self.vs[1]["mm__"] = """ApplyModel"""
        # paired with relation between match and apply models
        self.add_node()
        self.vs[2]["mm__"] = """paired_with"""
        self.vs[2]["attr1"] = """Father2Man"""

        # match class Parent node
        self.add_node()
        self.vs[3]["mm__"] = """Parent"""
        # match class Family node
        self.add_node()
        self.vs[4]["mm__"] = """Family"""
        # apply class Man node
        self.add_node()
        self.vs[5]["mm__"] = """Man"""

        # match association Parent--family-->Family node
        self.add_node()
        self.vs[6]["attr1"] = """family"""
        self.vs[6]["mm__"] = """directLink_S"""
        # match association Family--fathers-->Parent node
        self.add_node()
        self.vs[7]["attr1"] = """fathers"""
        self.vs[7]["mm__"] = """directLink_S"""

        # Add the edges
        self.add_edges([
                (0,3), # matchmodel -> match_class Parent
                (0,4), # matchmodel -> match_class Family
                (1,5), # applymodel -> apply_class Man
                (3,6), # match_class Parent -> association family
                (6,4), # association family  -> match_class Family
                (4,7), # match_class Family -> association fathers
                (7,3), # association fathers  -> match_class Parent
                (0,2), # matchmodel -> pairedwith
                (2,1) # pairedwith -> applyModel
    ])

        # Equations for node attributes
        self["equations"] = [((5,'fullName'),('concat',((3,'firstName'),(4,'lastName')))), ]
```

Figure 5.8 – The Himesis file for the *Father2Man* rule.

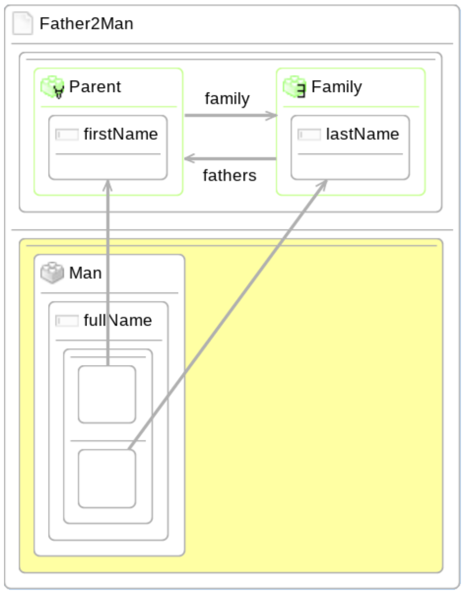


Figure 5.9 – A graphical representation of the *Father2Man* rule.

Lines 31-38 create the nodes to represent associations in the graph. As these associations are typed, they must be created as nodes in Himesis so that their type may be stored and matched by the match algorithm. Note that the *attr1* attribute stores the type of the association, while the *mm__* attribute denotes that these are typed as *directLinks* for the *S*ource component of the rule.

The edges for this graph are added in Lines 40-51 as source/target pairs between nodes. Edges are created between the match model and the match elements, and the apply model and the apply elements, to denote which elements are in which component.

Note that associations are present as two graph edges. The first edge is from the source node to the association node, while the second edge connects the association node to the target node. This representation of nodes and edges has complexity consequences for matching algorithms. In particular, it is quite expensive to iterate through the list of edges and find the nodes which are adjacent to a particular node through an association. This is further discussed in Section 6.2 when we present the matching algorithm used in the contract prover.

Finally, Line 54 defines the equations for this rule. These equations are our implementation of defining how attributes are set on the produced nodes when the rule is executed. In this example, the *fullName* attribute of node *5* (the *Man* element) is set as the concatenation of the *firstName* of the *Parent* element and the *lastName* of the *Family* element. These equations are resolved during the matching and rewriting steps, as discussed in Section 5.3.4.

Note that we have made changes to the Himesis format during the course of our research, as compared to the versions used in [122] and [145]. These changes were made for speed and space optimization.

In particular, we noted that attribute storage was very space-inefficient, in that if one node had a value for an attribute, then all nodes would reserve space for that attribute. For example, if the value *Father2Man* was stored in the attribute *name* for only one node in the graph, all other nodes would reserve space for the attribute *name*[2]. This was extremely space-intensive, and had severe performance repercussions in certain cases.

Our work-around for this issue is to assign new names to the attributes, such that they follow the form *attr1*, *attr2*, etc. As many nodes only need one or two parameters, a minimum number of these empty attributes is created. This technique is employed in Listing 5.8 on page 178 on Lines 19, 33, and 37.

5.3.2 T-Core

For increased usability and performance, we determined that the SyVOLT contract prover must be built upon a framework to provide a precise level of control in model manipulation. To this end, we have built our contract prover upon the T-Core model transformation library [145], which was built as a means to rapidly build high-level model transformation languages. As such, it offers a collection of model

2. The value is set to *None*, which is the Python equivalent of *null*

transformation primitives such as the *Matcher*, *PreConditionPattern*, *Iterator*, and *Rewriter* constructs.

The work of Syriani [145] discusses how these primitive constructs can be combined with a control logic language to successfully de-construct and reconstruct several model transformation languages. In fact, the DSLTrans language itself has been implemented using T-Core primitives [91].

We provide a brief discussion here about the T-Core primitives used within our approach. Section 5.3.3 will discuss how these primitives are built from the artifacts produced by the frontends, while Section 5.3.4 and Section 5.3.5 discuss their use within the prover.

5.3.2.1 T-Core Primitives

The essential unit of information in T-Core is the *Packet*. These *Packets* hold the result of matching, rewriting, or iterating. As such, they are composed of a graph, as well as extra information such as the *PreConditionPattern* primitive that was matched on the graph, along with the locations of the matches.

The *Matcher* primitive in T-Core represents the matching concept, as it stores a *PreConditionPattern* graph to be matched onto a *Packet*. This *PreConditionPattern* is composed of a graph to match as well as an optional *negative application condition* (NAC) graph. As in the semantics of DSLTrans, if the NAC matches then no match can be found (Section 3.2.2).

When a packet is passed to the *Matcher*, the *Matcher* will determine the positions of any *PreConditionPatterns*, and whether the NAC would also match. For further control, it is also possible to specify the number of matchings allowed by the *Matcher*. In the contract prover, the *Matcher* primitive will be used to find if rules can be combined with path conditions, or if the contract can match over the path condition.

Once the *Matcher* has determined the set of mappings of the *PreConditionPattern* onto the packet graph, an *Iterator* primitive then selects one of the matches found by the matcher. This selection of the next match can be ordered or random, non-deterministic or deterministic, but must be reproducible.

It is possible to successively refine matches by employing another *Matcher* primitive after the first, or instead to use a *Rewriter* primitive to transform the matched elements. The *Rewriter* primitive stores a *PostConditionPattern* to apply on each match located in the packet. In our contract prover, this is used to add rule elements into the path condition, as discussed in Section 4.2.1 and Section 5.3.4.

The chaining together of *Matcher* primitives thus produces a *Query* on the model to find elements (as in Figure 5.10a), while the combinations of *Matcher* and *Rewriter* primitives forms an *Application* of a transformation rule (Figure 5.10b). Note that these examples are written in a Python implementation of T-Core, termed *Py-T-Core*. This is discussed in the next section.

```
class Query(Composer):
 def __init__(self, LHS):
   super(Query, self).__init__()
   # To find 1 match
   self.M = Matcher(condition=LHS, max=1)
   # To select the only match
   self.I = Iterator(max_iterations=1)
 def packet_in(self, packet):
   self.is_success = False
   packet = self.M.packet_in(packet)
   if not self.M.is_success: return packet
   packet = self.I.packet_in(packet)
   if not self.I.is_success: return packet
   self.is_success = True
   return packet
```

(a) T-Core *query*.

```
class ARule(Composer):
  def __init__(self, LHS, RHS):
    super(ARule, self).__init__()
    self.M = Matcher(condition=LHS, max=1)
    self.I = Iterator(max_iterations=1)
    self.W = Rewriter(condition=RHS)
  def packet_in(self, packet):
    self.is_success = False
    packet = self.M.packet_in(packet)
    if not self.M.is_success: return packet
    packet = self.I.packet_in(packet)
    if not self.I.is_success: return packet
    packet = self.W.packet_in(packet)
    if not self.W.is_success: return packet
    self.is_success = True
    return packet
```

(b) T-Core *rule* matching and rewriting once.

Figure 5.10 – Examples of T-Core usage (from [145]).

In Figure 5.10a, the *Query* is a subclass of the *Composer* primitive, which contains other primitives and thus gives hierarchical structure to the transformation [145]. In the query's initialization method, the *Matcher M* is created with the input *LHS* set as the pattern. As well, it is specified that only one match should be made of this pattern. Next the *Iterator I* is created with only one iteration (as there will be only one match).

The *Query* is activated when the *packetIn* method is called, with the target graph stored in the packet argument. First M will store the found match in the packet. Then the iterator I will select the only match.

Figure 5.10b shows how the addition of the *Rewriter W* to a query creates an *Application* of a rule. The rewriter is created from the input *RHS* to the rule. When the rule is activated, the *Matcher* and *Iterator* select one match location, before *R* rewrites the matched location with the *RHS*.

5.3.2.2 Py-T-Core

Note that the T-Core primitives shown above do not define an implementation. In fact, it is possible to use them outside of the graph-transformation paradigm of our contract prover [145]. Our contract prover makes use of the implementation of T-Core in Python, termed *Py-T-Core* [142].

Py-T-Core implements T-Core within a graph transformation paradigm operating on the Himesis format. Therefore, the matching algorithm used is a sub-graph isomorphism, and the rewriting operation is the single push-out approach. This single push-out approach is similar to the double-pushout approach (Section 3.2.1.1), but does not include the interface graph in between the matching and rewriter graphs. Therefore, there can be no issues with *dangling edges*, where a vertex that is the source or target of an edge is deleted and the edge is not.

5.3.3 PyRamify

For the path condition generation steps detailed in Section 5.3.4, our contract prover requires T-Core *Matchers* and *Rewriters* for each rule in the transformation. Specifically, we require a *Matcher* to determine whether a rule can combine with a path condition, as discussed in Section 4.2.

For example, we will require these *Matchers* when we are examining whether the rule *Father2Man* (described in Figure 5.7 and Listing 5.8) can combine with a path condition during path condition construction. To accomplish this, the *PyRamify* operation constructs a *Matcher* which contains a *PreConditionPattern*, where the elements in that pattern can match over *Father* and *Family* elements. Thus, the pattern needs to be able to match over concepts from both the input language and output language.

The PyRAMify operation therefore automates the creation of patterns to match over rules, given those rules as input. This allows the SyVOLT prover to directly manipulate rules as graphs through matching and rewriting.
First, this section details the operations involved in the *RAMification* technique, which creates a pattern language from meta-models. Then an explanatory figure is provided to motivate how the operations we perform are a superset of those in RAMification. Finally, we will detail the *Matchers* and *Rewriters* produced by PyRamify which are necessary for contract prover operation.

5.3.3.1 RAMification

The RAMification technique takes as input a meta-model, and creates a pattern language to match over that meta-model. It can also be used to create property languages in other modelling-based approaches, such as *ProMoBox* [103]. The term *RAMification* [85] stands for the *relaxation*, *augmentation*, and *modification* operations performed on the meta-model to produce the pattern language.

— **Relaxation:** A pattern may not entirely conform to its language due to multiplicity or abstractness constraints. For example, an abstract class in the meta-model may be required in the pattern language, and therefore these constraints must be relaxed. Kuehne *et al.* [85] suggest the removal of the lower bound on multiplicities on elements, allowing abstract classes to be made concrete in the pattern, and the removal of any constraints which check for completeness.

— **Augmentation:** In the augmentation step, any extra attributes or links required for the transformation purpose are added to the pattern language.

— **Modification:** The modification step differs on how the RAMification technique is being used to create a pattern language for the negative application condition/left-hand side of a rule, or the right-hand side of a rule. For the first case, each element attribute in the original language is replaced by a constraint on that attribute in the pattern language. In the second case, the attributes are replaced by actions, which can change the attributes of the matched-over element when the rule applies. An example is described below.

The RAMification approach may also involve changing the *concrete syntax* (visual representation) of the meta-model. Note that this is not applicable to PyRamify, as SyVOLT does not display or manipulate the concrete syntax of languages.

PyRamify Details. In our application of RAMification, we augment each element in the pattern language with two new attributes: a flag to indicate modification by a rewriting operation, and a label for identifying nodes in the match mapping.

Note that Kuehne *et al.* [85] suggest storing sub-typing information on each node in the pattern language, such as storing that a *Person* matches over both a *Parent* and a *Child* on every *Person* node. This is very inefficient in terms of storage. In our current implementation, this information is stored once as an attribute at the graph level for each instantiated pattern where it can be accessed by the matching algorithm.

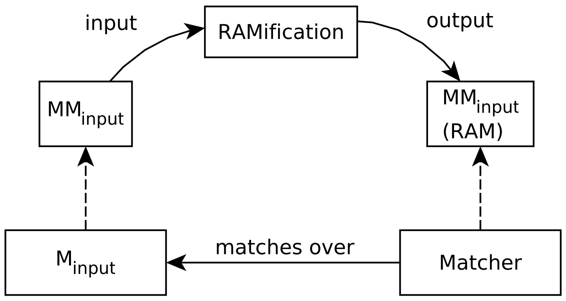
As well, during the modification stage of RAMification, for the negative application and pre-condition pattern language, PyRamify adds the string *MT_pre__* to each attribute for elements in the language so that these attribute are marked as constraints on that attribute.

For example, consider a *directLink_S* association with an attribute *name* and a value of *fathers*, then the respective *directLink_S* node in the *PreConditionPattern* will have an attribute *MT_pre__name* with a value of 'return attr_value == *fathers*'. During matching, this code will check the constraint on the node it is matching over.

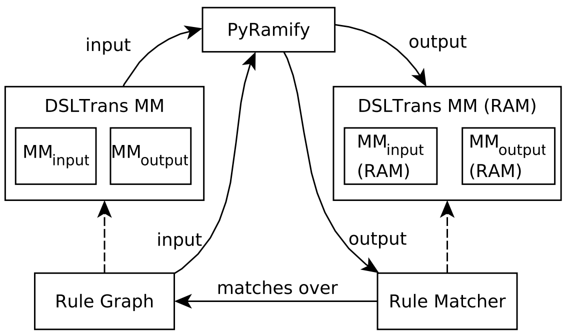
5.3.3.2 Explanatory Figure

The RAMification technique takes as input a meta-model, and produces a pattern language to match onto that meta-model. In contrast, the PyRamify technique takes a rule in the DSLTrans language, and produces a matcher which matches over that particular rule. This section includes figures to illustrate these concepts.

In Figure 5.11a, the usual case for the RAMification technique is shown which is described in the original RAMification work [85]. On the left, there is an input model which is typed by a meta-model. The RAMification technique takes the meta-model and produces a pattern language through the *relaxation*, *augmentation*, and *modification* operations. This pattern language created through RAMification then types any number of matchers, which represent the left-hand side of rules in that they are able to match over models of the original language.



(a) Creating a pattern language from a meta-model.



(b) Creating a pattern language to match over DSLTrans rules.

Figure 5.11 – Examples of RAMification for creating pattern languages.

Figure 5.11b displays how we apply the RAMification procedure to artefacts to create matchers for them. As seen in Section 5.3.1, the contract prover operates on DSLTrans rules, which are represented as a model in the Himesis formalism. The path condition generation process requires matchers which can match over these rules and path conditions to resolve dependencies (as in Section 5.3.4).

The RAMification procedure is therefore performed on the meta-model which types a DSLTrans rule. Note that this meta-model also composes the input and output meta-models, as these elements appear in the rule. For example, in the *Father2Man* rule which is presented in Section 5.3.1, the *MatchModel* and *ApplyModel* originate from the DSLTrans language, the *Father* and *Family* types are found in the *Families* meta-model, and the *Man* type comes from the *Persons* meta-model (Section 2.2.1).

The RAMification procedure takes this composed meta-model, and creates the equivalent pattern language. We can then construct a matcher which matches over a DSLTrans rule represented as a model. In our example, the matcher would be able to successfully match over the *Father2Man* rule graph shown in Section 5.3.1.

5.3.3.3 Matchers and Rewriters Created

In our SyVOLT tool, the PyRamify component takes as input the rule graph and rule meta-model, and directly produces the T-Core *Matchers* and *Rewriters* which are typed by the pattern language. This automatic RAMification allows for the prover to operate directly on the rule graphs, the contract graphs, and the input and output meta-models, without requiring further artefacts or user intervention.

There are a number of *Matchers* and *Rewriters* required for the path condition construction and contract proof algorithms described in the next sections. We present an overview of them in Table 5.1, including their purpose and the elements they contain.

Backward Link Matcher *Contains:*	Determines if a path condition contains the necessary dependencies for a rule. The backward links in a rule and the connected elements.
Rule Matcher *Contains:*	Determines if a rule must or may combine with a path condition. Also used to determine if a rule will match over another rule, which is termed *subsumption* in Section 4.2.3.2. All elements in the rule's matcher, as defined in Section 3.1.4.
Rule Rewriter *Contains:*	The rewriting graph used to add elements when combining the rule with a path condition. The elements in the rule's rewriter as defined in Section 3.1.4. Note the addition of traceability links.
Contract Matchers *Contains:*	The matchers for the contract's pre-condition and post-condition for contract proof. The elements for these matchers are discussed in Section 4.4.2.1.

Table 5.1 – The T-Core *Matchers* and *Rewriters* created by the PyRamify component.

Note that these *Matchers* and *Rewriters* operate on typed graphs represented in the Himesis format. As well, we have modified them to such that they can match and rewrite equations present on the graphs as seen in a rule in Listing 5.8. This is further discussed in Section 6.2.2 on page 201.

5.3.4 Path Condition Creation

As discussed in Section 4.2, our symbolic execution process will generate all path conditions which represent the infinite set of all possible transformation executions.

Once the required T-Core *Matchers* and *Rewriters* have been produced by the PyRamify component, the prover moves on to the path condition generation step. This section will discuss the broad steps of the implementation including how the *Matchers* and *Rewriters* are used.

Note that path condition generation is a complex task. As such, the algorithm in this component has been implemented as a Python script that is responsible for scheduling all sequences of matching and rewriting in the right order. This programmatic level of granularity of manipulation has allowed us to optimize the path condition generation process, for example through parallelization as described in Section 6.3.

As in Section 4.2, the path condition construction algorithm treats each layer of the transformation in turn. Each path condition from the path condition set from the previous layer is examined to see if it can be combined with the rules in the current layer.

This combination step is based on the matching results for the *Backward Link Matcher* and *Rule Matcher* created by the PyRamify component. If the *Backward Link Matcher* for a rule fails to match on the path condition, then the rule's dependencies cannot be met. If the *Rule Matcher* successfully matches then the rule *must* combine, otherwise it only *may* combine. If the rule can combine, then the *Rule Rewriter* operates on the path condition to add the rule elements.

During path condition construction, we also perform a number of algorithmic optimizations as discussed in Section 6.3. These include the parallelization of path conditions amongst different threads, the slicing of the transformation for a particular contract, and the removal of invalid path conditions.

5.3.5 Contract Proof

Once the path conditions are generated, the prover will determine which path conditions satisfy or do not satisfy the contracts to be verified. The contract proof component of the prover requires two inputs: the path conditions set built by the path condition generator and the *Contract Matchers* for the contract's pre-condition and post-condition,

This contract proof component is also implemented as a Python script that manipulates the *Contract Matchers* to determine if they match over each path condition in the path condition set. The component checks to see if the pre-condition matcher succeeds, and if it does, whether the post-condition matcher also succeeds. As each path condition is independent of the others, this operation has also been parallelized amongst different threads as discussed in Section 6.3.

Based on these results, the path conditions are divided into sets as described in Section 4.4.3. The prover then performs analysis of the results to help the user understand the results (Section 6.4.3). These results are then returned to the user through the front-end which began the process, allowing the user to remain in one tool.

5.4 Conclusion

This chapter has provided a detailed discussion of the SyVOLT tool, which is our implementation of the path condition construction and contract proving process.

SyVOLT is based on the principles of model-driven development as it operates on formalisms and models in a modular design. This is described in Section 5.1 where we present a diagram of the work-flow of the tool as a *Formalism Transformation Graph and Process Model*.

The methods for creating DSLTrans transformations and contracts to be verified have been presented in Section 5.2. This includes our higher-order transformation for converting ATL transformations to the DSLTrans language, as well as plug-ins for the Eclipse and MPS environments.

Section 5.3 has also presented a few details on the implementation of the contract prover as Python scripts, including a presentation of a DSLTrans rule in the Himesis format (Section 5.3.1). These Himesis graphs are manipulated by T-Core *Matchers* and *Rewriters* during the path condition construction and contract proof algorithms.

As well, we have also described the PyRamify component in Section 5.3.3. This component operates on DSLTrans rules to automatically create the *Matchers* and *Rewriters* able to match over and manipulate those rules.

Chapter 6
SyVOLT Algorithms

Our symbolic execution approach has its origins in the thesis of Barroca [19]. While that formulation is valid and complete, due to the explicit nature of its symbolic execution it is not feasible for the scale of industrial transformations we wish to verify.

For example, in Barroca's thesis, over 11,000 path conditions are generated for a transformation containing nine rules. As a rough comparison, the current implementation of the prover generates only 10 path conditions for the *GM-to-AUTOSAR* transformation (Section 7.2) which is of comparable size (number of rules and number of elements).

Our technique tackles the critical scalability problem by limiting the multiplicity of rule application within path conditions (Section 4.2), and relying on the abstraction relation to avoid creating explicit path conditions of all possible input/output model pairs(Section 4.1).

An important algorithmic innovation in this thesis is the detailing of the typed graph split morphism which was presented in Section 2.3.2.2 and is critical to our abstraction relation and contract proving approach. As shown in Section 4.4.2, this morphism is required to let a contract match over a path condition which represents multiple input-output models. An algorithm and complexity analysis will be provided in Section 6.2, along with experimental results of the new graph matcher compared to a standard isomorphic matcher.

Another contribution of this work is the presentation in Section 6.3 of further algorithmic techniques to reduce the time and space requirements of the prover to improve the scalability of the approach. This includes standard software techniques such as parallelization and compression, but also transformation-focused algorithmic techniques such as slicing the transformation based on the contract to be proved, and pruning invalid path conditions based on the transformation meta-model. Complexity analyses and experimental results are provided for each of these techniques.

Finally, Section 6.4 discusses our efforts to assist the user when proving contracts. We provide three techniques to a) ensure that all rules are symbolically executed enough times to be able to satisfy the contract, b) ensure that during path condition construction that all rules are represented in a path condition, and c) assist the user in understanding how rule interactions satisfy or don't satisfy a contract.

Therefore, this chapter answers our last research question: *What are techniques for improving the scalability of the verification tool to allow for larger transformations to be verified?*

6.1 Experimental Conditions

Throughout this chapter and the next, we conduct experiments to highlight the optimizations performed in our prover and demonstrate the feasibility of our case studies.

Our experimental machine is a (relatively old) desktop machine, with 6 gigabytes of memory and an Intel Core i7 CPU 920 at 2.67GHz with 8 threads on 4 cores. The operating system is 17.10 Xubuntu, and the Python version used to run the experiments is Python 3.6. This is the primary machine used for experiments, unless indicated otherwise.

A more powerful computer was required to validate the *mbeddr* transformation, due to the transformation's large size (Section 7.5). This experimental machine was a supercomputer located at McMaster University, and is an Intel Xeon CPU E7-4870 at 2.40GHz with 32 threads on 32 cores and 1 terabyte of memory. The Python version is 3.4.

When obtaining numeric results, efforts were taken to reduce variance. Each experiment was run five times, the highest and lowest values were removed, and the average was taken. The variance of all path condition generation experiments was below 1%, except for the *mbeddr* case study where variance was below 3%. For contract proof, variance was highest at around 5% for the *Families-to-Persons* transformation, as a relatively large number of contracts were proven. The other case studies have a variance of around 1% or less for contract proof. As well, the original match set generation results in Table 6.1 on page 206 for the *mbeddr* case study have an extremely large variance, as remarked upon in that section.

The *mbeddr* transformation

Note that we have not been able to build the path condition set for all 49 rules of the *mbeddr* transformation (Section 7.5 on page 270) on our experimental desktop machine, due to a lack of disk space and memory. Therefore, throughout the following sections, we refer to each version of *mbeddr* as $mbeddr_i$ where the i represents the number of rules in the transformation. Thus, $mbeddr_{22}$ refers to the sliced version[1] of the transformation, $mbeddr_{34}$ the largest portion of the transformation that we were able to verify on our desktop experimental machine, and $mbeddr_{46}$ refers to the full transformation.

6.2 Split Morphism Matching Algorithm

This section discusses our split morphism matching algorithm, which produces a mapping from the nodes and edges of the pattern graph to the nodes and edges of the target graph. In this section, the mapping will be termed a *match set.*

1. See Section 6.3.2 on page 222 for discussion of the slicing algorithm.

As discussed in previous chapters (Section 4.2.2, Section 4.1.5.3, Section 4.4.2.3) our symbolic execution and contract proving approaches requires a matching algorithm that is different from a standard graph isomorphism. The typed graph split morphism is defined in Definition 5 on page 37, where sub-morphisms are used to "split" the pattern graph over the target graph.

As an example, this morphism is employed in Section 4.4.2.3 on page 133 as a way to match contracts over path conditions, as the path conditions are composed of individual rules which may or may not have overlapping elements. An alternative approach is an expensive "disambiguation" operation as seen in the thesis of Barroca [19], where explicit path conditions are created that contain every possible overlapping of elements in the input-output models. However, this approach is infeasible for medium or large transformations.

We note that our algorithm performs quite well on reasonably-sized transformations, as shown in the results throughout this chapter. As well, we conducted experiments on a database of synthetic graphs to determine the performance relative to an isomorphic matcher. These results are presented in Section 6.2.5.

As a caveat to these results, indirect link matching and the semantics of the *Exists* element are not currently implemented. While we have an algorithm for performing these matchings, the algorithmic complexity precludes their implementation at this time.

Our explanation of the *split morphism matching algorithm* is split into two steps. For both of these steps, a complexity analysis is performed of their current implementation.

The first step of the matching algorithm involves iterating through both the pattern and target graphs and generating the candidate lists where pattern elements may match over target elements. This is presented in pseudo-code in Algorithm 2 on page 197, while Algorithm 5 showcases the algorithm for resolving the transitive closure of a graph. As well, any typing issues and attribute constraints between the pattern and target elements must be resolved, as presented in Algorithm 3 on page 198 and Algorithm 4 on page 198.

The second step of the matching algorithm is the creation of the match set which forms the actual mapping between pattern and target elements, which is presented in pseudo-code in Algorithm 6.

Algorithm 2 – The split morphism match algorithm.

Input: Two graphs representing the pattern graph and target graph.
Output: Mappings of pattern nodes to target nodes.

```
 1: function SPLITMATCHALG(patt, target)
 2:     candidates ← $\emptyset$
    Match all isolated nodes
 3:     for each isoNode $\in$ patt.isolated do
 4:         for each node $\in$ target.match do
 5:             if MATCHNODES(isoNode, node) then
 6:                 Add node to candidates[isoNode]
 7:         if candidates[isoNode] = $\emptyset$ then
 8:             return $\emptyset$                                          ▷ does not match
    Match all backward links
 9:     for each pattBackwardLink $\in$ patt.backward do
10:         for each traceLink $\in$ target.trace do
                ▷ Note that backward links match over trace links
11:             if MATCHLINKS(pattBackwardLink, traceLink) then
12:                 Add traceLink to candidates[pattBackwardLink]
13:         if candidates[pattBackwardLink]= $\emptyset$ then
14:             return $\emptyset$                                          ▷ does not match
    Match all direct links
15:     for each pattDirectLink $\in$ patt.direct do
16:         for each directLink $\in$ target.direct do
17:             if MATCHLINKS(pattDirectLink, directLink) then
18:                 Add directLink to candidates[pattDirectLink]
19:         if candidates[pattDirectLink] = $\emptyset$ then
20:             return $\emptyset$                                          ▷ does not match
    Match all indirect links
21:     closureMx ← CREATECLOSUREMX(target)                                 ▷ See Algorithm 5
22:     for each pattIndirectLink $\in$ patt.indirect do
23:         for each targetSrcNode $\in$ target.match do
                ▷ Test if the source node in the pattern matches the target source node
24:             if MATCHNODES(pattIndirectLink.source, targetSrcNode) then
25:                 for each targetDestNode $\in$ closureMx[targetSrcNode] do
                        ▷ Check each connected node to see if the pattern target node matches
26:                     if MATCHNODES(pattIndirectLink.dest, targetDestNode) then
27:                         Add (targetSrcNode, ‘indirect’, targetDestNode) to candidates[pattIndirectLink]
28:         if candidates[pattIndirectLink] = $\emptyset$ then
29:             return $\emptyset$                                          ▷ does not match
    Generate match sets one-at-a-time
30:     for each mapping $\in$ GENERATEMAPPINGS(candidates) do
31:         yield mapping
```

6.2.1 Step 1 - Candidate Generation

The candidate generation step determines each possibility for how the elements in the pattern graph can match over the elements in the target graph. This selection of candidate matches for each pattern element is represented by the *candidates* map variable on Line 2 of Algorithm 2.

The algorithm begins by selecting candidates for each of the components in the pattern graph: isolated nodes (match nodes with no connected edges), backward links, direct links, and indirect links. Note that searching for candidates in a different order may influence the run-time of the matching algorithm, based on the elements in the pattern graph. For example, it may be preferable to search for the pattern graph's backward links before the isolated nodes depending on which set is smaller, as not finding any one of the components means no match can be found. We note that a static analysis may assist in providing heuristics on how to match a pattern graph most efficiently.

The candidate selection depends on Algorithm 3 and Algorithm 4, which are the functions for how individual nodes and links may match over each other.

Algorithm 3 – The function for matching a pattern node against a target node.

```
Input: Two nodes
Output: Boolean result whether the pattern node matches over the target node
1: function matchNodes(pattNode, targetNode)
2:     typingMatch ← resolveTyping(pattNode, targetNode)
3:     attributeMatch ← resolveAttributes(pattNode, targetNode)
4:     return typingMatch $\wedge$ attributeMatch
```

Algorithm 4 – The function for matching a pattern link against a target link.

```
Input: Two links (tuples of two nodes and an association)
Output: Boolean result whether the pattern link matches over the target link
1: function matchLinks(pattLink, targetLink)
2:     sourceMatch ← matchNodes(pattLink.source, targetLink.source)
3:     targetMatch ← matchNodes(pattLink.target, targetLink.target)
4:     isBackwardMatch ← pattLink.assoc.type == 'backward' $\wedge$ targetLink.assoc.type == 'trace'
5:     assocMatch ← isBackwardMatch $\vee$ matchNodes(pattLink.assoc, targetLink.assoc)
```

```
6:      return sourceMatch $\wedge$ targetMatch $\wedge$ assocMatch
```

Isolated Nodes. First, the algorithm determines if the isolated nodes in the pattern graph can match over nodes in the target graph. These isolated nodes are any that are not connected to a link, and will therefore not be examined by matching the other components. The matching of the isolated nodes is presented as pseudo-code in Lines 3 to 8 in Algorithm 2. A call is made to the *matchNodes* function (Algorithm 3 on the preceding page) to determine if the nodes can match, based on the meta-model types of both the pattern and target nodes, as well as any attributes defined on those nodes. This resolution is discussed in Section 6.2.2.

If an isolated node in the pattern graph cannot match over any target graph node, then the matching fails and an empty match set is returned from the algorithm. Otherwise, the candidate nodes are placed in the set for each isolated pattern node.

Backward Links and Direct Links. A similar process to the isolated nodes is then performed for the backward links in the pattern (Lines 9 to 14) and the direct links (Lines 14 to 20). Note that instead of singular nodes being matched, instead the entire link is matched. The *matchLinks* function (Algorithm 4 on the previous page) is used to match all three elements in the link (the end nodes and the association node). Note that a special type allowance is made such that backward links match over trace links.

Indirect Links. The candidate selection for the indirect links in the pattern graph is more complicated than the direct or backward links. The semantics of DSLTrans is that indirect links must match over the transitive closure of the graph (Section 3.2.4). Therefore, to represent this matching in path condition construction, the indirect links in the matching algorithm must also match over the transitive closure.

The call to the function in the matching algorithm is in Line 21 in Algorithm 2. The pseudo-code for the transitive closure function is shown in Algorithm 5, and is based on the Floyd-Warshall algorithm [52], a dynamic-programming problem where intermediate solutions are used to build the full solution.

Algorithm 5 – The transitive closure function algorithm.

```
Input: A graph
Output: A boolean matrix of the transitive closure of the graph. i.e. if the matrix at [i, j] is true, then it is possible
    to navigate from node $i$ to node $j$
 1: function createClosureMx(graph)                    ▷ Based upon the Floyd–Warshall algorithm [52]
 2:     n ← graph.size
 3:     mx ← a two-dimensional array of $n$ by $n$, filled with zeros
 4:     for each edge ∈ graph.edges do
            ▷ Mark all existing edges
 5:         mx[edge.source][edge.target] = 1
 6:     for $k \leftarrow 0, n$ do
 7:         for $i \leftarrow 0, n$ do
 8:             for $j \leftarrow 0, n$ do
 9:                 mx[i, j] ← mx[i,j] ∨ ( mx[i, k] ∧ mx[k, j] )
10:     return mx
```

The intuition with this algorithm is to look at an intermediate node k between the start node i and the end node j. As k increases, the algorithm successively considers paths that can pass through nodes up to index k. This is reflected in Line 9 in Algorithm 5, where node i is connected to node j if there already exists a path, or if there is a path from node i to node k, and from node k to node j.

Note that this transitive closure creation algorithm is very expensive. For the Floyd-Warshall algorithm, the processing of the edges and three nested for-loops means it is $\theta(|E| + |V|^3)$ in time complexity. Note that the results of this function may be cached to prevent expensive recalculation.

Once the transitive closure matrix is built, the process to find candidate matches is similar to that of direct links and backward links. The difference is that instead of picking a link to examine, the algorithm looks at each node and the nodes that can be reached through the transitive closure. This is seen on Line 23 of Algorithm 2 on page 197 where each target node is examined, and if the pattern source node matches over that one, then the destination node is selected from the transitive closure graph for that target node on Line 25.

6.2.2 Typing and Attribute Resolution

Our matching algorithm operates on typed attributed graphs, so the type and attribute characteristics of the nodes must be examined to determine if two nodes match.

Type Matching. Each node in our typed graphs has a meta-model type. For example, Section 5.3.1 shows how each node and association in the Himesis graph has its type stored as the *mm__* attribute.

Therefore, during the matching of nodes during the matching algorithm (Algorithm 3 on page 198) it must be determined whether the pattern node type is the same or is a super-type of the pattern node type. If not, then these nodes cannot be matched together. For example, *Member* nodes in a *Families-to-Persons* transformation contract (Section 2.2.1 on page 23) match over *Member*, *Parent*, and *Child* nodes in a path condition.

Attribute Matching. As well as resolving typing information, there may be attribute constraints on the graph to resolve.

For example, consider a rule that defines an association with the *fathers* type. In the matcher created for this rule by the PyRamify component (Section 5.3.3 on page 183), the respective pattern node will contain an attribute named *MT_pre__type* with a value of *fathers*. This indicates that there is a constraint on the *type* attribute of the target node to be found. During the attribute resolution step, this constraint must be taken into account, and the value on the target node checked. If the value is not correct, then the pattern node and target node cannot match together.

Attribute values can also be represented by equations on the graph. For example, the *Father2Man* rule (Figure 5.9 on page 179 in Section 5.3.1) defines equations on the *fullName* attribute of the elements in the pattern, where the *fullName* of the produced *Man* element is a concatenation of the *lastName* of the *Family* element and the *firstName* of the *Parent* element. This equation is represented by the statement ((5,*fullName*), (*concat*, ((3,*firstName*),(4,*lastName*)))), which means that the *fullName* of node number five is assigned to the concatenation of the values on node three and four.

As explained in Section 3.2.3 on page 59, we handle the resolution and matching of values using a simple equation evaluator. The String concatenations and attribute references in the equations are resolved, and these resolved values are compared with the attribute values stored in the candidate node, as well as any equations in the target graph for that node.

For example, a pattern equations may specifies the *wildcard* token concatenated with a String value. This wildcard matches over any number of characters in a String, as introduced in Section 2.2 on page 22. This is useful for matching attributes that end or begin with a certain String. In the attribute solver, the presence of a wildcard means that a simple regular expression is built to match the pattern expression over the target Strings. The success or failure of the regular expression then determines whether the two nodes match.

6.2.3 Step 2 - Match Set Creation

The first matching step (Section 6.2.1 on page 198) will create a set of candidate matches for each element in the pattern graph. The *match set* is then created to map each node in the pattern graph to a specific node in the target graph.

The complexity with the match set generation algorithm is in ensuring consistency within the match set. For the typed graph split morphism, we must ensure that while a pattern node can match to multiple target nodes (the 'splitting'), a target node cannot match to multiple pattern nodes.

Note that in the mapping produced, a pattern graph node will be matched with a single target graph node. This restriction is for the rewriting procedure, which may attach edges to the target node. It is not correct to rewrite the graph for each target node matched by the pattern node, nor is it correct to 'glue' the various target nodes together. Each node is left separate as required by the abstraction relation (Section 4.1 on page 82).

Pseudo-code for a recursive version of this match set generation algorithm is seen in Algorithm 6 and Algorithm 7 on the following page. The entry function GENERATEMAPPINGS in Algorithm 6 transforms the dictionary of candidates matches into a list before the mappings from the recursive match set generation algorithm are successively returned.

When the recursive algorithm GENERATEMAPPINGSR in Algorithm 7 on the following page is first called, the match set is empty, the reverse match set is empty, and there is a list of tuples which contain pattern elements and their candidate matches. The intuition of the algorithm is that this list will shrink one element at a time as the mapping is created from the pattern elements to the candidate elements. The reverse mapping is stored to ensure that a target element is not mapped to more than one pattern element.

Algorithm 6 – The entry function GENERATEMAPPINGS for generating a match set.

```
Input: A dictionary of nodes and links to their candidate matches
Output: A mapping of pattern node IDs to target node IDs
1: function GENERATEMAPPINGS(candidates)
2:     candidateList ← an empty list
        ▷ Transform the mapping into a list
3:     for each key, values ∈ candidates do
4:         Add (key, values) to candidateList
        ▷ Call the recursive function
5:     for each mapping ∈ GENERATEMAPPINGSR(∅, ∅, candidateList) do
6:         yield mapping
```

The base case of the recursive algorithm is on Line 2 of Algorithm 7 on the next page, where the list of pattern and candidate elements has been reduced to zero. In this case, an empty mapping is returned.

In the recursive case, we begin by taking the first tuple of the candidate list using the *head* operation. This tuple consists of the pattern element (Line 4) and a list of target elements to be matched onto by the pattern element (Line 5).

The target elements are iterated through on Line 6. Before placing the target's IDs into the match set, it is important to make a copy of both the match set and

Algorithm 7 – The recursive algorithm GENERATEMAPPINGSR for generating a match set.

```
Input: A mapping of pattern node IDs to target node IDs, a mapping of target node IDs to pattern node IDs, and
    the candidate links to consider
Output: A mapping of pattern node IDs to target node IDs
 1: function generateMappingsR(matchSet, reverseMatchSet, candidates)
 2:     if candidates.size == 0 then                                                    ▷ Base case:
 3:         return ∅
 4:     pattElement ← head(candidates)[0]
 5:     targetElements ← head(candidates)[1]
 6:     for each targetElement ∈ targetElements do
            ▷ Make copies of sets to avoid aliasing
 7:         matchSetCopy ← copy(matchSet)
 8:         reverseMatchSetCopy ← copy(reverseMatchSet)
 9:         skip ← false

            ▷ Elements could be a node, or three nodes in a link
10:         for $i$ ← 0, pattElements.size do
11:             if (targetElement[i].ID ∈ reverseMatchSet ∧
12:                 reverseMatchSet[targetElement[i].ID] ≠ pattElement[i].ID) then
                    ▷ Fail, as the target element is already matched to by another pattern element
13:                 skip ← true
14:                 break
                ▷ Store the mapping, and the reverse mapping
15:             matchSetCopy[pattElement[i].ID] ← targetElement[i].ID
16:             reverseMatchSetCopy[targetElement[i].ID] ← pattElement[i].ID

            ▷ Check for the consistency of Exists elements
17:         for $exists_1$, $exists_2$ ∈, existsElementPairs do
18:             isCombining ← matchSetCopy[$exists_1$.ID] == matchSetCopy[$exists_2$.ID]
19:             if (existsMustCombine($exists_1$, $exists_2$) ∧¬ isCombining ) ∨
20:                 (existsMustNotCombine($exists_1$, $exists_2$) ∧ isCombining) then
21:                 skip ← true
22:                 break

23:         if skip then                        ▷ There was a failure, consider the next target element
24:             continue
            ▷ Get other mappings recursively
25:         otherCandidates ← tail(candidates)
26:         for each rest ∈ generateMappingsR(matchSetCopy, reverseMatchSetCopy, otherCandidates) do
                ▷ copy to avoid aliasing, update the mapping with more values, and yield it as a valid mapping
27:             mapping ← copy(matchSetCopy)
28:             mapping.update(rest)
29:             yield mapping
```

reverse match set, to avoid aliasing problems where all iterations and recursions of the algorithm operate on the same sets.

Then, the for-loop on Line 10 will iterate from zero to the size of the pattern element. This is a detail to handle the fact that the elements may be individual nodes, or they may be links.

Line 12 is crucial to detect consistency in the mapping. If the target element's ID is already in the reverse match set, and is not the current pattern element's ID, then we fail this mapping. This means that another node in the pattern graph has already bound to this node in the target graph, and two pattern nodes cannot match over the same node in the target graph.

If the check for consistency passes, then the mapping and reverse mapping can be updated, as on Line 15. If not, the next target element is examined.

The presence of *Exists* elements in our pattern graph must also be taken into consideration during the consistency check. This pseudo-code is presented on Lines 17 to 22. Note that we assume the construction of a set that contains all pairs of elements that are the *Exists* elements from the pattern (if applicable). If the two elements are considered to *must combine* or *must not combine* (following the semantics defined in Section 3.2.6 on page 65) then the algorithm checks whether they will or will not match over the same target element, and fails the consistency check as needed.

Finally, the rest of the pattern elements must be given mappings recursively. The tail of the candidate list is provided by the *tail* operation on Line 25. Then, the for-loop on Line 26 will iterate through all the mappings given by passing the match set, reverse match set, and tail of the candidate list to another call of the recursive match set generation function.

Each mapping given by the recursive function will be merged (using the *update* function) with a copy of the current match set. This will be the mapping returned to a previous call of the recursive function.

The recursive match set generation algorithm therefore operates in a tree-like call pattern. The valid mappings for one combination of the pattern elements and target elements are selected, before being combined with all the possible mappings of the rest of the candidate list.

Comparison. The initial implementation for creating the match set depended on first generating all possible combinations of the mapping in a brute force fashion, then examining each mapping for consistency. This implementation was found to be too slow to our case studies, and we now employ the recursive algorithm described above. We provide this comparison as a justification for the recursive algorithm, and to highlight the impact of algorithm selection in graph matching.

Table 6.1 – Effects of changing match set generation algorithm

Transformation	**Match Set Creation Method**	**PC Build Time(s)**	**Contract Proof Time (s)**
Families-to-Persons	Generate all combos	7.73	17.74
	Recursive solver	7.17	14.58
UMLToKiltera	Generate all combos	∼2	>25 min.
	Recursive solver	1.82	4.17
$mbeddr_{22}$	Generate all combos	2.14	81.84
	Recursive solver	2.16	0.55

The results for both the original match set generation algorithm and the recursive version are seen in Table 6.1. Note that the recursive version is dramatically faster for the *UML-To-Kiltera* and $mbeddr_{22}$ transformations. In fact, the experiment was cut short for the UMLToKiltera transformation with the original algorithm, due to excessive delay.

As described in Section 6.1 on page 193, the variance of our experiments is around 3%. However, for this experiment, the variance for the $mbeddr_{22}$ transformation under the original match set generation algorithm was an extreme value of 1000 seconds, demonstrating the instability of that algorithm.

6.2.4 Complexity Analysis

This section will discuss the complexity of each major component of the split morphism matching algorithm, and present a brief comparison to two other matching algorithms.

For this section, let the pattern graph be denoted as p, and the target graph be denoted as t. For example, the number of vertices in the pattern graph will be denoted $|V_p|$, while the number of edges is denoted $|E_p|$.

Typing Resolution. $\mathcal{O}(1)$ - The matching of nodes must take into account the types of the nodes in order to perform sub-type matching, as discussed in Section 6.2.2 on page 201. Note that while this matching could depend on the size of the meta-models for the graphs, pre-processing can reduce this to a constant time operation by creating a lookup table.

Attribute Resolution. $\mathcal{O}(M)$ - Also discussed in Section 6.2.2 on page 201 is the resolution of equations and attribute values on the pattern and target graph.

It is difficult to provide a complexity bound for this component, as the number of attributes for each node in the pattern graph, and the complexity of each equation may vary widely. For example, an equation may include many concatenations and wildcards, which requires the use of a regular expression to solve.

We therefore abstract the resolution of attributes with a general term M. Note that in our case studies (Section 7), the vertices contain only a few attributes, and the equations are composed of less than a dozen terms. Thus M should be a small constant. However, note that the below complexity arguments each involve M. It should therefore be highly optimized in graph matching algorithms.

Isolated Elements. $\mathcal{O}\big((|V_p| \cdot |V_t|) \cdot M\big)$ - Each isolated node in the pattern graph needs to be considered against every node in the target graph, while the M term represents the actual matching of the nodes.

Backward/Direct Links. $\mathcal{O}\big((|E_p| \cdot |E_t|) \cdot M\big)$ - Each direct and backward link in the pattern graph is matched against their counterparts in the target graph to determine candidates.

Transitive Closure Mx.. $\mathcal{O}(|V|^3)$ - As mentioned in Section 6.2.1 on page 198, the complexity of the transitive closure matrix generation step is quite high as all paths in the graph must be considered.

Indirect Links. $\mathcal{O}\big((|E_p| \cdot |V_t|^2) \cdot M\big)$ - For each indirect link in the pattern graph, a source node must be found in the target graph. Then, all connected nodes in the target graph must be examined to determine if they are a suitable destination node.

Match Set Generation. In the recursive match set creation step, the execution forms a branching-tree pattern as each pattern element is given a mapping to a target graph element. For each level of the tree, each target element may be considered, which is on the order of $|V_t| + |E_t|$ elements in the worst case. As there are $|V_p| + |E_p|$ pattern elements (and therefore levels of the tree), the complexity is $\mathcal{O}\big((|V_t| + |E_t|)^{|V_p|+|E_p|}\big)$.

With the addition of *Exists* element resolution, we must also add a comparison of potentially every pattern element versus every other pattern element. Therefore, the complexity rises to $\mathcal{O}\big((|V_t| + |E_t| + |V_p|^2)^{|V_p|+|E_p|}\big)$.

This complexity seems intractable. However, whole branches of the tree may be pruned when inconsistent mappings are detected. As well, the entire tree does not needed to be searched, as graph-matching often requires only one consistent mapping.

6.2.4.1 Conclusion and Comparison

To produce a rough time complexity for the entire matching algorithm, we sum and simplify the above complexities in Equation 6.1. Note that for readability, we assume that the pattern and target graph are of similar size [2].

$$\mathcal{O}\big(|V|^3 + M \cdot (|E|^2 + 2 \cdot |E| \cdot |V|^2) + (|E| + |V| + |V|^2)^{|V|+|E|}\big) \tag{6.1}$$

Equation 6.1 shows that the matching of two graphs is a very expensive computational algorithm. However, the results throughout this chapter show that it is suitable for the transformation sizes encountered in our case studies.

Comparison to Other Algorithms. One of the most-popular sub-graph isomorphism matching algorithms in the literature is the VF2 algorithm [44]. This algorithm constructs a search-tree of the pattern graph. Nodes are evaluated for compatibility, and the algorithm will backtrack on failure. The algorithm has a best case time complexity of $\theta(|V|^2)$, and a worst-case time complexity of $\theta(|V|! \cdot |V|)$ [44]. A new version of the algorithm, termed VF3, improves upon the speed of VF2 but not the complexity [40].

The thesis of Syriani [142] discusses the extension of VF2 with a new node compatibility strategy to form a new sub-graph isomorphism matching algorithm to reside within the Himesis graph operations kernel [122]. This improvement is slower than VF2 for small graphs, but is much faster for large graphs, with a complexity of $\mathcal{O}(|V|!)$.

Note that these two algorithms focus on finding a sub-graph isomorphic mapping. As discussed in Section 4.1 and Section 4.4, the nature of the abstraction relation means that we must employ a non-isomorphic morphism.

2. Note that this assumption is most likely violated for most model transformation applications, where the pattern graph is much smaller than the target graph. Therefore, this complexity is an over-approximation.

The text of Provost [122] mentions that the Himesis matcher may be made non-injective, with a complexity of $\mathcal{O}(|V_t|^{|V_p|})$. This avenue will be explored in our future development of the split morphism matcher.

6.2.5 Experimental Results

In this section, we present performance results from matching synthetic graphs using both our split morphism matching algorithm and the isomorphic matching algorithm from the Himesis kernel.

The experiments we performed were intended to answer the following research questions. First, *how long does the split morphism matching algorithm take to match pairs of pattern and target graphs*? Second, *how does this performance compare to the isomorphic matcher which has been created as part of the Himesis kernel* [122, 142]?

Graph Characteristics. The synthetic graphs used are sourced from the MIVIA database [48]. This database provides graphs with different characteristics such as isomorphism or sub-graph isomorphism between pairs of graphs. The graph structure is in two, three, and four dimensions with up to 1276 nodes, which is over four times larger than our sizeable *mbeddr* case study (Section 7.5 on page 270). As each category contains 100 pairs of graphs, this database offers hundreds of tests of our matching algorithm.

For our experiments, we divided the graphs up into four categories [3].

— *iso*: Pairs of isomorphic graphs

— *si2*: Sub-graph isomorphism where the sub-graph is 20% of the target graph

— *si4*: Sub-graph isomorphism where the sub-graph is 40% of the target graph

— *si6*: Sub-graph isomorphism where the sub-graph is 60% of the target graph

Within these four categories, graphs also varied in three other characteristics:

— *Dimension*: non-mesh, 2D mesh, 3D mesh, 4D mesh

— *Random Edges Added*:

3. See `http://mivia.unisa.it/datasets/graph-database/arg-database/documentation/` for further information.

— For the non-mesh graphs, 0.1, 0.05, or 0.01 chance of two nodes being joined

— For the mesh graphs, 0, 20%, 40%, or 60% extra edges

Results. Figure 6.1 contains two scatter plots for our matching experiments performed on our supercomputer machine as described in Section 6.1. On the x-axis is the number of nodes for each graph in the pair of graphs being matched, while the y-axis is a logarithmic scale for number of seconds for matching. Therefore each point represents the time it took to match each pair of graphs in the database.

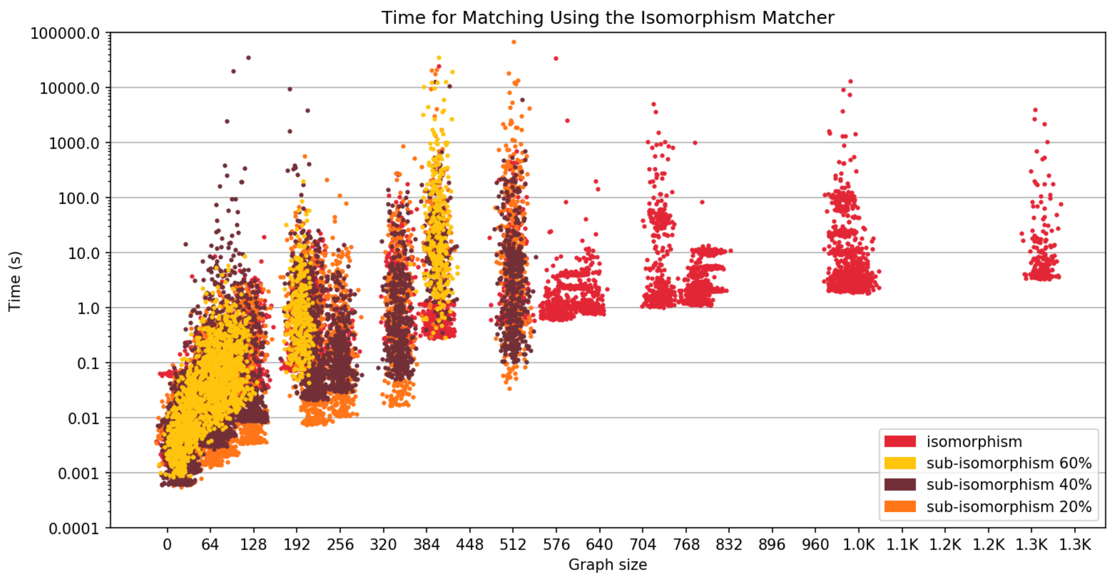
The different sets of data represent the coverage of the pattern graph over the target graph. For example, the pattern graph may be a sub-morphism of 20, 40 or 60 percent of the target graph [4], or even a total isomorphism. As seen in the figure, requiring larger coverage leads to an increase in matching time, with the isomorphism requirement taking the longest for both matching algorithms.

Note that some artistic liberties have been taken. In particular, the x-coordinate of each point was perturbed using a normal function to create an column effect for each graph size.

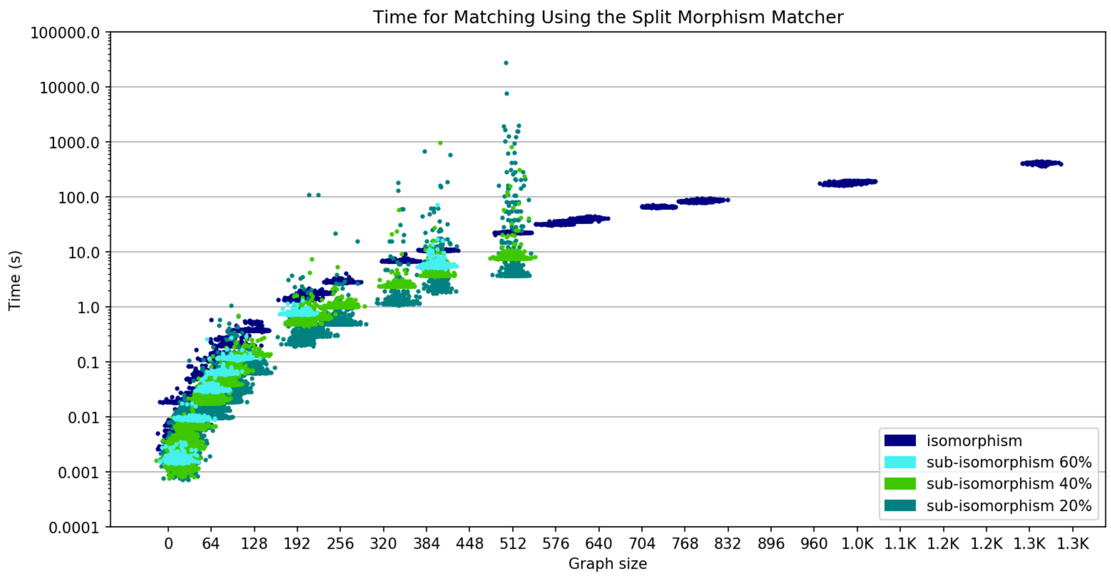
As well, we must mention that this should not be a direct comparison between the two matching algorithms. The split morphism allows for much greater flexibility in how the pattern graph matches on the target graph because the structure is not strictly maintained. In contrast, the isomorphism matcher must ensure that the exact pattern graph must be found in the target graph. Therefore, these results are presented only to provide a rough comparison for scalability to large graphs.

Figure 6.1 demonstrates interesting comparisons between the isomorphic matcher results and the split morphism matcher. The split matcher has a much smaller variance among graphs of the same size, especially for the isomorphic pairs of graphs. This is probably due to the backtracking performed by the isomorphism matcher to ensure that the structure of the pattern graph is found in the target graph. As

4. Note that this may not be representative of real-world pattern sizes, where the pattern graph is often smaller than the target graph by orders of magnitude.



(a) The isomorphism matcher from the Himesis kernel.



(b) The split morphism matching algorithm.

Figure 6.1 – Comparing the matching times for two matching algorithms.

mentioned, the split morphism matcher does not have this constraint, and therefore an earlier candidate set generated may be returned as a match.

Other than the variability, we note that the split morphism matcher tends to require much longer than the isomorphism matcher. For example, the larger graphs tend to take around 10 seconds to match in the isomorphism matcher, while the split morphism matcher takes 100 to 1000 seconds. This result is expected, as the split morphism matcher has not undergone extensive optimization. For instance, it may be possible to integrate more of the structural or typing information of the graph into the matcher to reduce matching time [167].

6.3 Algorithmic Optimizations

This section will detail three major algorithmic optimizations which were implemented in the SyVOLT prover. These optimizations were performed to decrease the time taken to prove each of the case studies (Chapter 7), and enable efficient verification for sizeable transformations such as the *mbeddr* transformation (Section 7.5) on our experimental desktop machine.

The first optimization discussed is the *parallelization* of the path condition generation and the contract proving steps. This is a fairly straightforward implementation of parallelization as the work on each path condition is independent. However, there are interesting points to mention regarding the load-balancing of path conditions amongst the threads, and how the time taken by the prover scales with the number of threads used.

The second optimization, *slicing* the transformation, chooses the minimum set of rules necessary for the contract to be proved. This can result in a reduction of the number of rules in the transformation, leading to shorter path condition generation and contract proof time as fewer path conditions are created for the sliced transformation.

The final algorithmic optimization presented introduces the notion of *pruning* invalid path conditions. In this optimization, we check if a produced path condition is missing containment links which cannot be satisfied by further rules in the transformation. If this is true, then the path condition can be discarded as it represents invalid output models in the transformation. This has a significant effect on the number of path conditions built by our generation technique.

For each of these optimizations, an overview will be provided along with the algorithm and a complexity analysis. Experimental results are also shown to demonstrate the effects of each optimization.

6.3.1 Parallelization

A classic software optimization effort is to perform parallelization on any computational algorithms. For the SyVOLT tool, this parallelization is performed on the path condition construction (Section 5.3.4 on page 188) and contract proving (Section 5.3.5 on page 189) algorithms. These algorithms were selected for parallelization as they almost totally determine execution time at 96 percent of the total running time. As well, they operate on individual path conditions so that parallelization was relatively easy to implement.

Therefore for parallelization, we create a number of *worker threads* to carry out operations, while coordination is performed by the *main thread*. In the sections below, we discuss the specifics about what is passed to and returned from each worker thread.

6.3.1.1 Path Condition Generation

Section 4.2 presents an overview of the algorithm for generating path conditions. In that section, the key operation is that each path condition in the path condition set is combined with the rules for a layer to produce further path conditions.

Therefore to make the path condition generation process parallel, we create a number of worker threads which each take a portion of the path condition set for a layer. Each worker can then operate on one path condition at a time to combine that path condition with the rules. The new path conditions created are collected by the worker, and these new path conditions are returned to the main thread to be merged for the next layer or contract proof. This is sketched in Figure 6.2.

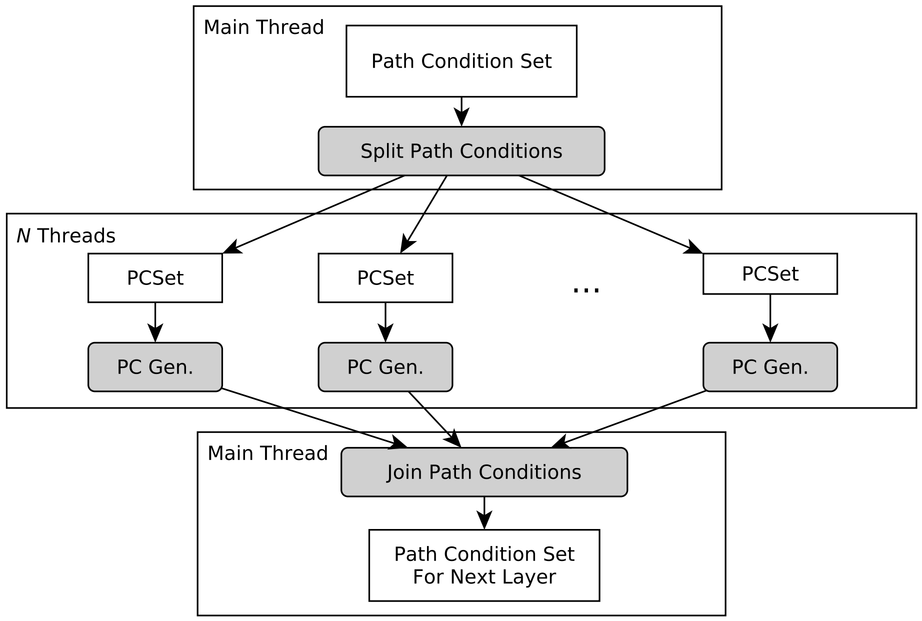


Figure 6.2 – Parallelization process for the path condition generation algorithm.

In Figure 6.2, the top box represents the main thread which partitions the set of path conditions for a layer into smaller pieces for each of the workers threads. These worker threads are represented by the components in the middle of the figure. Finally, the bottom box represents the collection of the new path conditions generated by each worker thread, where this joining creates the set of path conditions for the next layer.

Load Balancing. Two load-balancing schemes were devised to determine if there was a benefit to balance the portion of path conditions distributed to each worker, such that each worker did an equal amount of work.

The first scheme (pseudo-) randomly *shuffles* the path conditions amongst the workers. The second scheme counts the number of nodes in each path condition [5], and attempts to balance the total number of nodes per worker approximately equal by performing *bin-packing.*

The results for these two schemes are presented in Section 6.3.1.3 on the following page. As a preview, the results indicate that the bin-packing scheme is not significantly faster.

6.3.1.2 Contract Proof

As seen in Section 4.4 and Section 5.3.5, the contract proof algorithm requires a set of path conditions and the contract(s) to be proven on them. The result of this algorithm are the set of path conditions which fail to satisfy the contract, and the set that does satisfy the contract. This information will then be reported as the result of the contract proving process, and used for further contract debugging (Section 6.4).

To make this operation parallel, we again create a number of worker threads to match the contracts against each path condition. This process is sketched in Figure 6.3 on the next page. Each contract prover thread is given the contracts to be proved before it begins waiting on a queue from the main thread to the worker thread. Path conditions are placed one-at-a-time into the queue by the main thread represented in the top box of Figure 6.3. At the same time, the worker threads select path conditions one-at-a-time from the queue and prove each contract.

The path conditions which satisfy or do not satisfy the different contracts are recorded by each worker thread, and this information is passed back to the main thread to be presented to the user, as represented on the bottom side of the figure.

The intention for this queue-based scheme was to load-balance the path conditions amongst the various threads without explicit division of the set. However, as shown in

5. In our implementation, the number of nodes is stored in the path condition's name. Therefore, the path condition does not need to be loaded or de-serialized, making this load-balancing scheme take a fraction of a second.

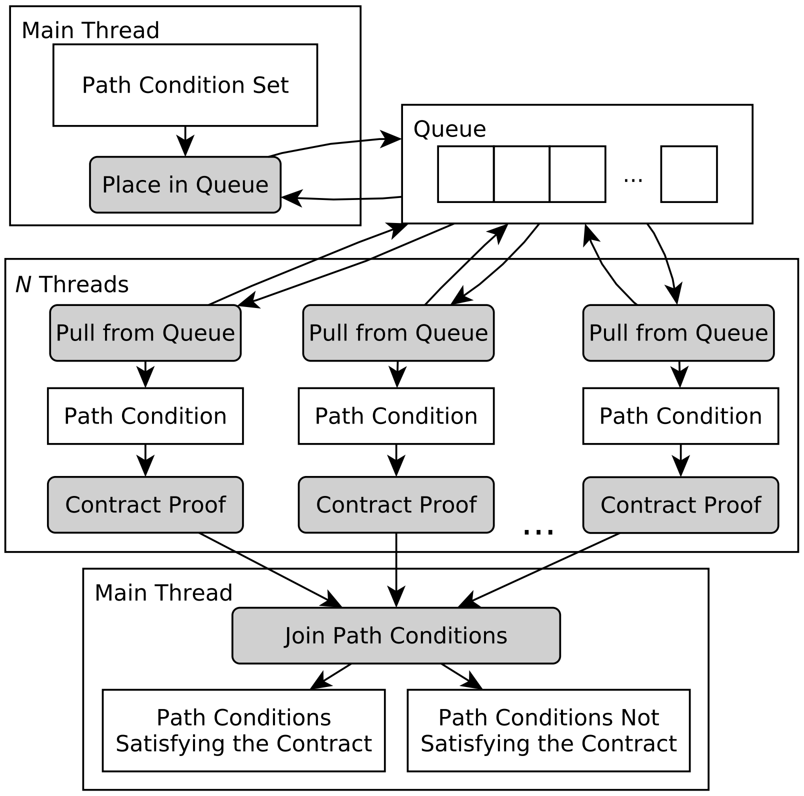


Figure 6.3 – Parallelization process for the contract proof algorithm.

Figure 6.4 on page 220 the communication overhead for passing each path condition may be detrimental when many worker threads are used.

6.3.1.3 Results

The effects of parallelization are seen in Table 6.2 for five experiments, *UML-to-Kiltera*, *Families-to-Persons*, and three versions of the *mbeddr* transformation (see Section 7.5). Note that the experiments for the full *mbeddr* transformation (denoted $mbeddr_{46}$) were run on the supercomputer, the details of which are presented in Section 6.1 on page 193. As mentioned in Section 6.1 on page 193, the variance of these results is below three percent.

The results in Table 6.2 show the difference in path condition generation and contract proof times when only one thread is used, and when eight threads are utilized. The speedup is displayed in brackets next to the timing results. We see that parallelization amongst the eight threads on the desktop machine offers a significant speed-up for each component, though this result is well below the theoretical eight times speed-up for a perfect parallelization. This result may be due to the architecture of our desktop machine. As stated in Section 6.1, the desktop machine has 8 threads divided amongst 4 cores. It may be that this sharing of cores leads to interference (cache utilization, heat/power issues, etc.) that do not allow for the potential eight times speed-up to be reached. As well, we theorize that the increase in number of threads has a negative effect on the queue load-balancing scheme used in contract proving.

Table 6.2 – Effects of parallelization with shuffling and bin-packing load-balancing strategies.

Transformation	**Parameters**	**Build PCs Time (s.)**	**Contract Proof Time (s.)**	**Memory (MB)**
UML-to-Kiltera	Non-parallel	5.81	14.62	82
645 path conditions	Shuffle	1.70 (3.4x)	4.31 (3.4x)	48
	Bin-packing	1.78 (3.3x)	4.32 (3.4x)	49
Families-to-Persons	Non-parallel	16.58	42.60	80
4916 path conditions	Shuffle	5.37 (3.1x)	15.75 (2.7x)	59
	Bin-packing	5.34 (3.1x)	16.12 (2.6x)	59
mbeddr$_{22}$	Non-parallel	5.70	1.75	96
1012 path conditions	Shuffle	1.77 (3.2x)	0.50 (3.5x)	57
	Bin-packing	1.88 (3.0x)	0.49 (3.6x)	57
mbeddr$_{34}$	Non-parallel	177.69	62.81	168
28018 path conditions	Shuffle	65.52 (2.7x)	22.80 (2.8x)	162
	Bin-packing	67.10 (2.6x)	23.33 (2.7x)	163
mbeddr$_{46}$	Non-parallel	15073.37	4430.14	1958
625486 path conditions	Shuffle	1297.11 (11.6x)	429.01 (10.3x)	1917
	Bin-packing	1002.05 (15.04)	430.99 (10.3x)	1949

Table 6.2 also shows the difference for the random shuffling load-balancing scheme versus the more sophisticated size-aware bin-packing scheme for path condition construction.

The results show that the more advanced scheme is only more effective than the randomized shuffle on a more powerful machine with more threads. Therefore, a random division may be effective enough to distribute the workload in most cases.

In terms of memory usage, Table 6.2 reports a rough calculation of resident memory usage, which we must point out is *per thread*. Therefore, the memory usage for the *Families-to-Persons* transformation goes from 80 MB to the single-threaded version to potentially 480 MB for the threaded version. This is due to each worker thread consuming at a minimum around 40 MB of memory due to the large structures involved in the proving process. Further investigations are ongoing to determine and reduce the amount of memory being accessed by each thread.

Parallelization on Desktop Machine. Figure 6.4 graphically represents the results of contract proof using the desktop machine on three of our case studies, when the number of threads is varied from one to eight.

For each of the *UML-to-Kiltera*, *Families-to-Persons*, and *mbeddr*$_{22}$ case studies, the time taken for path condition generation time (*PC Time*) and contract proving time (*Contract Time*) is graphed against an increasing number of threads.

For path condition generation, parallelization is beneficial when adding the first few threads. However, after the fourth thread, there seems to be no substantial benefit to adding threads. For contract proving time, there may even be a slight negative effect, especially for the *Families-to-Persons* transformation. Again, this may be due to the architecture of the desktop machine, and the queue-based parallelization for contract proving.

Parallelization on Supercomputer Machine. To address the potential issues in the desktop experiment, we then conducted another parallelization test on the supercomputer. The results for this test are shown in Figure 6.5 where we present

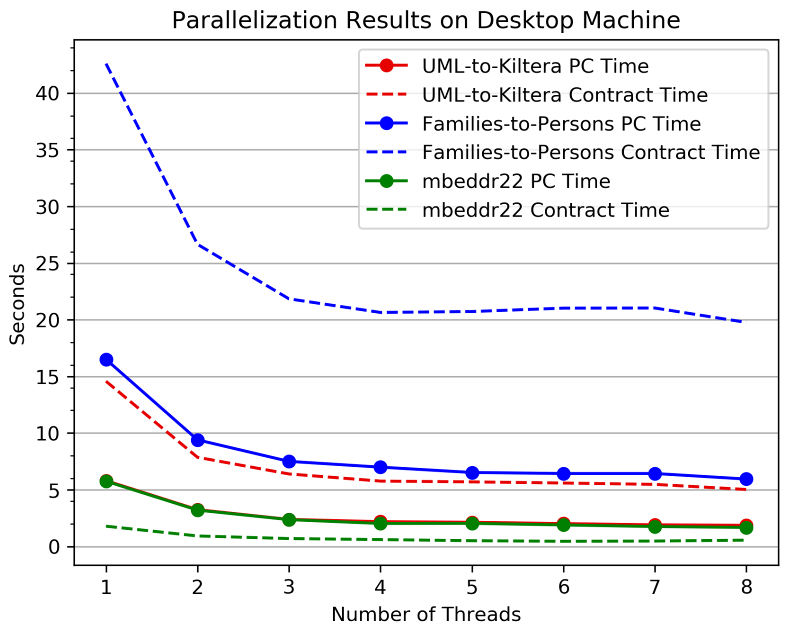


Figure 6.4 – Parallelization amongst threads on the desktop machine.

the time taken for path condition generation time (*PC Time*) and contract proving time (*Contract Time*) for the full $mbeddr_{46}$ transformation. The x-axis is the number of threads used, while the y-axis is the number of minutes taken.

As well, we add plots for the theoretical *ideal* 32x speedup if the parallelization process was perfect amongst the 32 threads.

The result shows that the path condition generation and contract proving steps has been successfully parallelized, decreasing the time taken from around 325 minutes for both path condition generation and contract proof to about 35 minutes. In fact, the parallelization curve follows very closely to the ideal curve, showing that this parallelization could potentially scale to even larger numbers of threads.

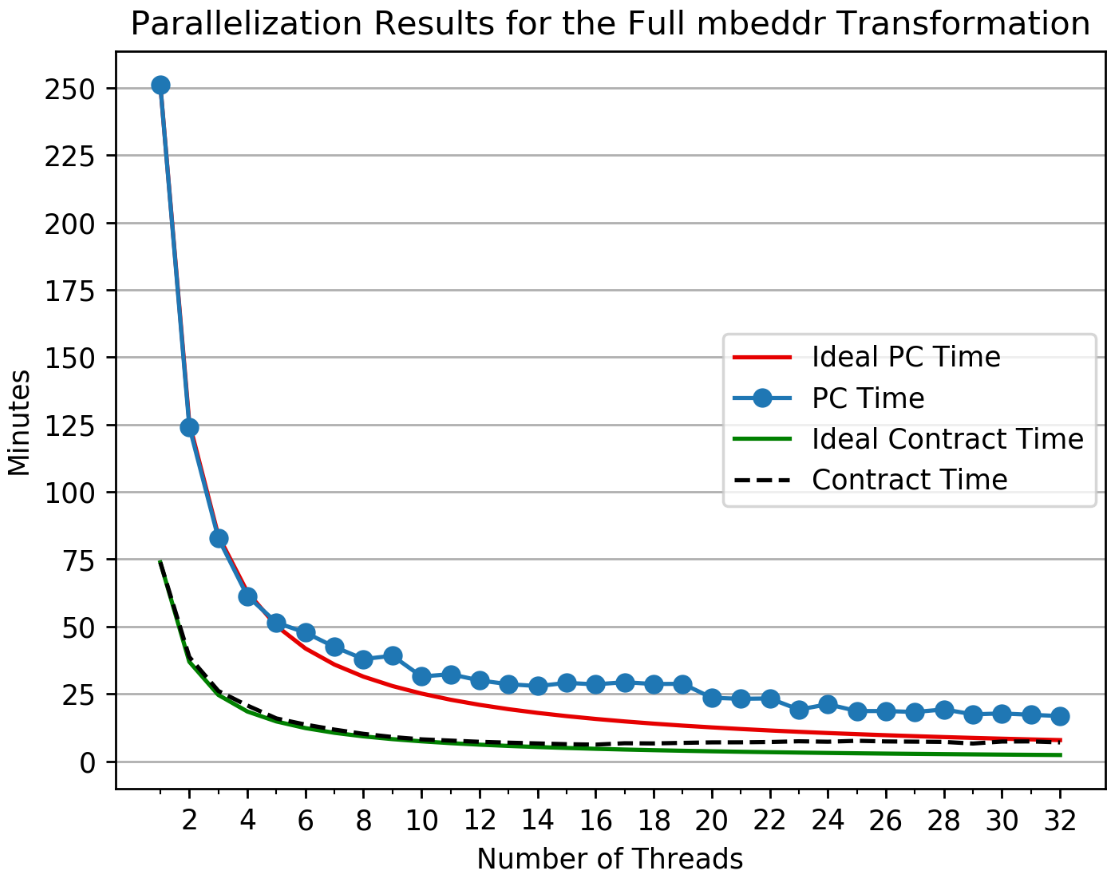


Figure 6.5 – Parallelization of the $mbeddr_{46}$ transformation on the supercomputer with up to 32 threads.

6.3.2 Slicing

The intention of the slicing algorithm is to partition the transformation for a given contract, such that only the rules needed to produce the correct verification result are symbolically executed. The selection of a minimal set of rules allows for a significant decrease in the amount of time required for the proving process, as seen in Table 6.3 on page 227.

Our slicing technique was first introduced in 2015 [112], and expanded on in 2016 [113]. In this thesis, we also add a concrete example of the rules produced by slicing the *Families-to-Persons* transformation, as well as a discussion on the complexity of the slicing algorithm.

6.3.2.1 Slicer Overview

There are three steps in our technique to slice the transformation for a contract. Here, we discuss these steps in the context of slicing the *Families-to-Persons* transformation for the *CountryCity* contract.

The *CountryCity* contract is displayed in Figure 6.6. The pre-condition for this contract is whether there is *Country* element to a *City* element, and the post-condition checks for the existence of an *Association* element.

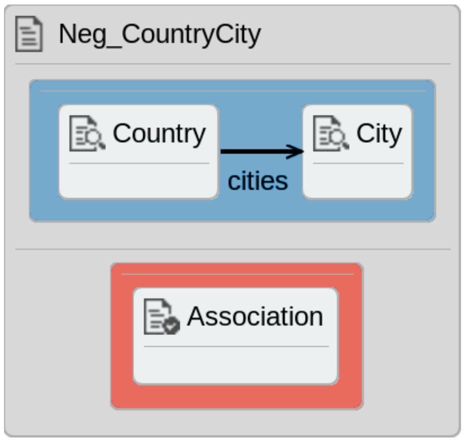


Figure 6.6 – *CountryCity* contract for the slicing process.

There are two steps to the slicing process: a) determine which rules produce elements in the contract and mark them for keeping, and b) recursively determine which rules the kept rules depend on. At the end of these steps, the rules marked as kept will form the sliced transformation.

Finding Required Rules. The first step in the slicing process performs an examination of all rules in the transformation to determine if any rule produces the elements in the contracts.

The rationale behind this step is that the path conditions which the contract matches over during contract proof are built by combining rules. Therefore, we must discover the rules which could build the necessary path conditions.

We thus match each node and link in the contract against every node and link in each rule in the transformation. If this match succeeds, then that rule may be essential to the proving of the contract by producing the required node or link, and the rule is retained in the sliced version of the transformation.

Note that if any node or link in the contract cannot be found in the full set of transformation rules, then the contract cannot match on any path condition produced by the proving algorithm. This indicates that either the contract or the transformation contains errors and must be fixed. An explanation of this analysis technique is presented in Section 6.4 on page 237.

As an example, we present the *CountryCity* contract in Figure 6.6 on the previous page and the four rules it depends upon from the *Families-to-Persons* transformation in Figure 6.7 on the following page. Note that the rules selected involve the *Country-City* association, as well as those that produce an *Association* from a *Country* or a *City* element.

Finding Rule Dependencies. The second step in the slicing procedure determines if those rules selected in the first step require nodes or links that are produced by earlier rules. Thus, the matching procedure is repeated again for each of these rules. This is an iterative process, as those earlier rules may require other rules, and so on.

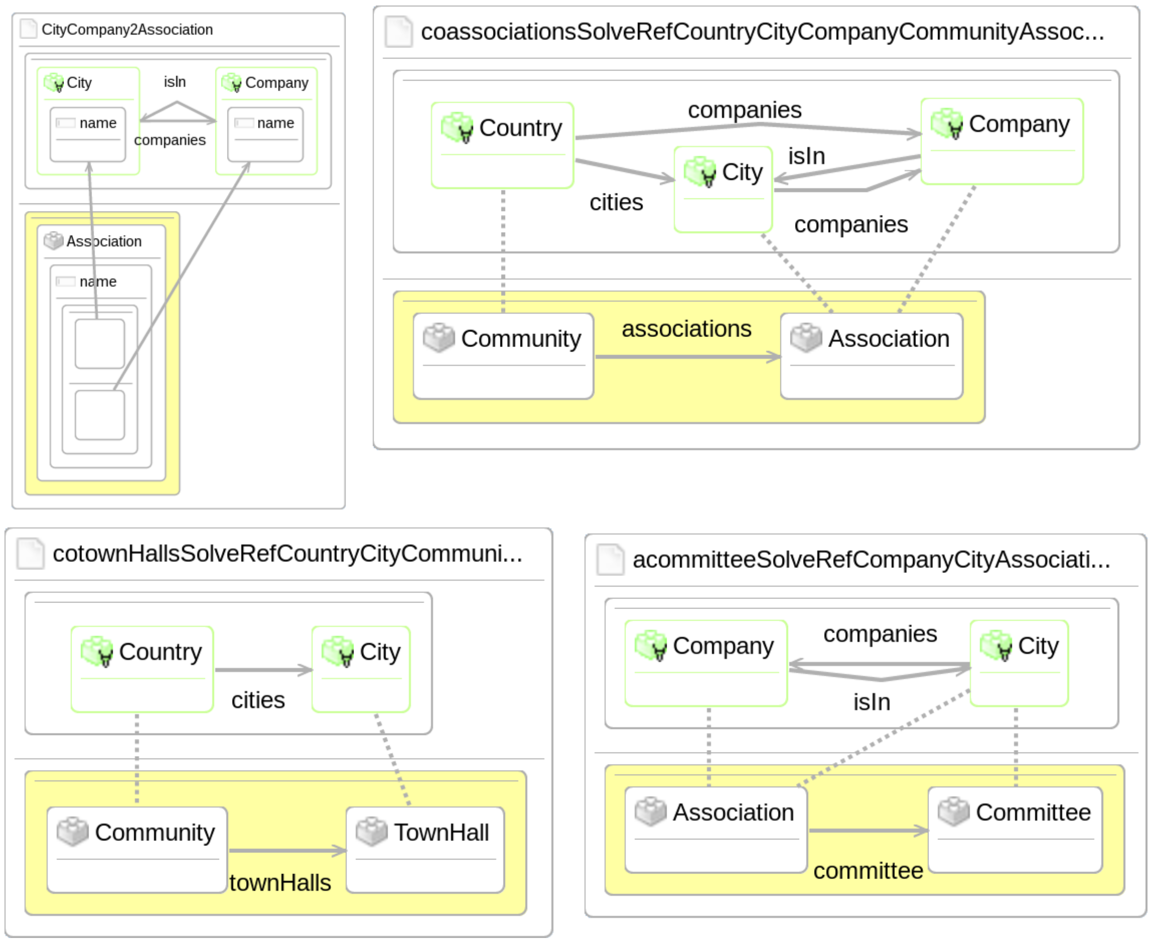


Figure 6.7 – Portion of rules selected for the *CountryCity* contract through slicing.

For example, the *cotownHalls* rule in Figure 6.7 requires a rule that produces a *Community* element from a *Country* element. This requirement is specified by the backward link.

This recursive step of finding rule dependencies brings the total number of rules required for the *CountryCity* contract up to eight, which is less than half of the 19 rules in the *Families-to-Persons* transformation. Table 6.4 on page 228 shows how this reduction in rules to be symbolically executed impacts the performance of the contract prover.

All those rules not required for the contract are removed from the transformation to be verified. Note that this new transformation may be smaller than the original, but this depends on the specific dependencies between the contract and the transformation rules. All rules may be potentially required for the contract to produce the correct verification results.

6.3.2.2 Complexity

The complexity for the slicing process primarily depends on the matching algorithm, and the number of rules in the transformation.

The rule matching step depends on the split morphism matching algorithm discussed in Section 6.2 on page 194. Rather than checking for the presence of all elements or links however, this matching in the slicing process is only concerned with a single element or link being found. Despite this, the complexity remains the same as matching the entire graph.

The contract is first matched against all rules to determine which rules the contract requires. Then, we repeat this rule matching procedure for every rule, against all other rules to determine the dependency tree. In the worst case, every rule in the transformation will be compared against every other rule.

The complexity of slicing is therefore $\mathcal{O}(X \cdot |Rules|^2)$, where X is the complexity of the matching algorithm (Section 6.2).

6.3.2.3 Results

This section discusses the impact of slicing the *Families-to-Persons* and *UML-to-Kiltera* transformations for each of their contracts. Each contract was proven on the full transformation (the 'original' version) and on the subset of the transformation returned by slicing for that contract (the 'sliced' version).

Note that the experiments below were performed with no pruning and no parallelization, to maximize the number of path conditions and time to generate them.

As well, note that the time taken to perform slicing itself was less than 0.05 seconds for all contracts.

UML-to-Kiltera Transformation Results. The results in Table 6.3 on the next page show the reduction in contract proving time for the *UML-to-Kiltera* transformation with slicing. The full transformation contains 17 rules, while the slicing operation generates subsets from 2 to 15 rules. These smaller sets of rules produce at most a third of the number of path conditions generated for the full transformation.

This reduction in path conditions generated is reflected in the decrease in path condition generation time and contract proof time. As well, the memory needed for the prover to operate is reduced due to fewer path condition produced. The memory requirements then become dominated by the minimum for each worker thread, around 40 MB.

Families-to-Persons Transformation Results. The results presented in Table 6.4 on page 228 show excellent results when the *Families-to-Persons* transformation is sliced for each contract. Note that the full version of the transformation produces 4916 path conditions, while the largest slice produces only 184 path conditions. The number of rules is reduced from 19 for the full transformation to a range of 5-11 rules for the sliced versions.

6.3.2.4 Slicing Discussion

The slicing technique is effective in producing reduced versions of the transformations specific to each contract. As can be seen in the tables in this section, a fraction of the full number of path conditions are produced, reducing both path condition construction time and contract proving time and reducing memory usage.

Table 6.5 shows a summary of how slicing affects the metrics of both transformations presented. For example, slicing the *UML-to-Kiltera* transformation reduces the path condition generation time by 73% in the worst case and by 98% in the best case.

Table 6.3 – Effect of slicing on contract proving time for the *UML-to-Kiltera* transformation
.

Contract Name	**Rules**	**Path Conds.**	**Path Cond. Build Time (s)**	**Contract Prove Time (s)**	**Memory (MB)**
Full	17	645	5.81	14.62	82
PP1	12	100	0.59	0.27	44
PP2	12	100	0.59	0.24	44
PP3	13	145	0.76	0.39	44
PP4	13	145	0.76	0.34	44
MM1	4	2	0.08	0.02	44
MM2	8	5	0.13	0.03	44
MM3	11	56	0.28	0.13	44
MM4	11	56	0.29	0.13	44
MM5	12	100	0.59	0.26	44
MM6	4	2	0.08	0.02	44
MM7	8	5	0.13	0.03	44
MM8	12	100	0.59	0.24	44
MM9	12	100	0.59	0.25	44
MM10	11	56	0.28	0.16	44
MM11	13	145	0.75	0.45	44
SS1	15	217	1.53	0.62	49

It is also important to note that the verifiability of the transformations has not been affected. We have ensured that the contract verification results are identical for both the normal and sliced versions for all of our case studies.

Table 6.4 – Effect of slicing on contract proving time for the *Families-to-Persons* transformation.

Contract Name	**Rules**	**Path Conds.**	**Path Cond. Build Time (s)**	**Contract Prove Time (s)**	**Memory (MB)**
Full	19	4916	16.58	42.60	80
CityCompany	8	43	0.23	0.07	43
CountryCity	6	10	0.10	0.03	43
SchoolOrdFac	5	17	0.13	0.04	43
DaughterMother	9	64	0.34	0.09	43
AssocCity	9	64	0.26	0.09	43
ChildSchool	5	17	0.13	0.04	43
FourMembers	9	64	0.34	0.09	43
MotherFather	9	64	0.35	0.10	43
ParentCompany	5	18	0.12	0.04	43
TownHallComm	11	184	0.59	0.27	43

Table 6.5 – Reduction in metrics caused by slicing.

Case Study	**Num. PCs Reduction**	**PC. Gen. Time Reduction**	**Contract Proof Time Reduction**	**Memory Reduction**
UML-to-Kiltera	66-99 %	73-98 %	95-99 %	40-46 %
Families-to-Persons	96-99 %	96-99 %	99 %	46 %

6.3.3 Pruning

The abstraction relation (Section 4.1) defines how our path conditions are abstractions over input-output models for the transformation. A logical conclusion is if invalid input-output models could be detected, then we could remove the path conditions which abstract over those input-output models.

The *pruning* operation is a validity check on path conditions based on the information in the transformation's output meta-model. In particular, we are interested in the presence or absence of *containment links* between elements, and whether a path condition satisfies these containment links or not.

For example, consider the *Families-to-Persons* output meta-model in Figure 6.8. The *Community* element contains *Person*, *Man* and *Woman* elements through the *persons* relation, which is marked as being a containment link by the black diamond. Therefore, all output models which contain a *Man* element must have that element attached to the *Community* element through this *persons* relation. Otherwise, the output model is invalid.

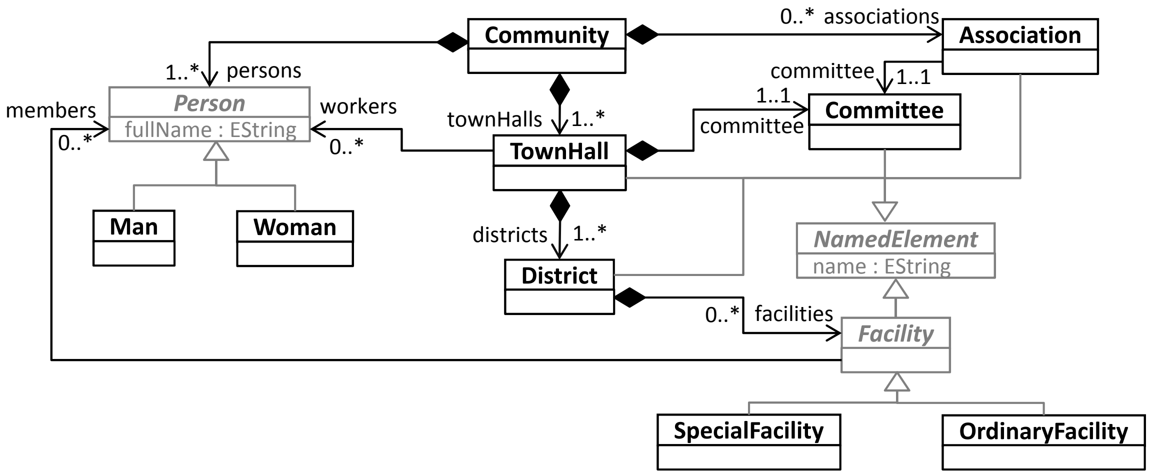


Figure 6.8 – *Persons* meta-model for the *Families-to-Persons* transformation.

Our insight is that the lack of these containment links (or lack of proper multiplicity) in a particular path condition implies the lack of this link in the abstracted input-output models. These path conditions therefore do not represent valid output models and can be removed from the set of path conditions for the transformation.

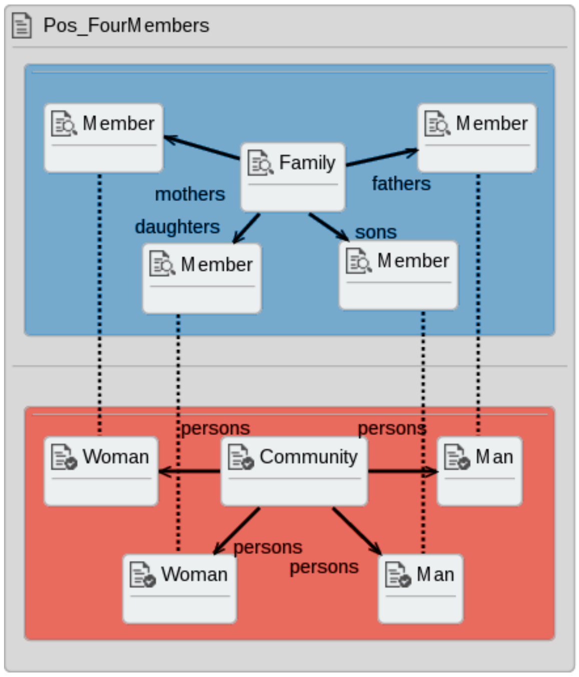
As with the slicing operation (Section 6.3.2), removing path conditions dramatically speeds up the creation of path conditions and contract proving. However this pruning operation occurs during path condition construction, as path conditions are examined and removed at the end of each layer of the path condition construction process.

Interaction with Contract Proving. Note that pruning may not be desired by the user, as they may be interested in examining the full range of possible rule combinations, not just those that will produce a valid output model. In fact, the pruning operation can prune away path conditions that would or would not satisfy contracts.

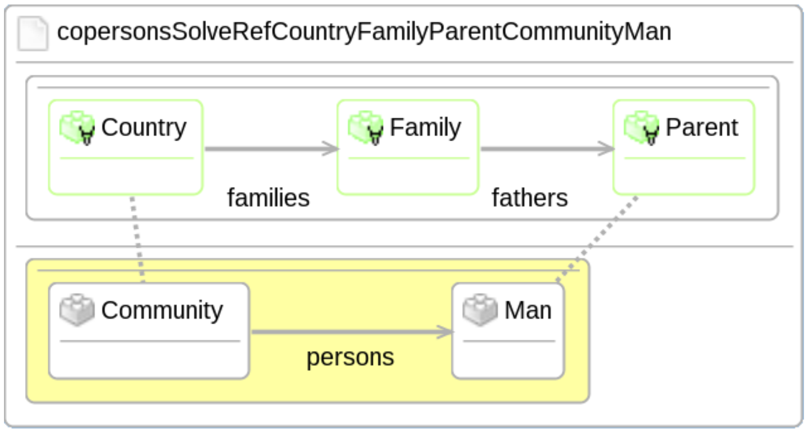
Note that the set of pruned path conditions is a subset of the full (unpruned) set. If a counter-example is found in the pruned set, then it will exist in the full set. However, the reverse is not true. If a counter-example exists in the full set, it will not exist in the pruned set if it is an invalid path condition.

The *FourMembers* contract for the *Families-to-Persons* transformation (seen in Figure 6.9a on the next page) is an example of a contract which produces different counter-examples whether pruning is enabled or disabled. When pruning is disabled, counter-examples to this contract are found which do not exist when pruning is enabled.

The reason for this difference in the counter-examples produced (indicated in Section 7.4.1 on page 263) is that the contract is checking in the pre-condition for the *Family* and *Member* elements, which by themselves do not enforce that the *persons* relation is constructed. This *persons* association is a containment link, and these associations are built in the four *copersons...* rules of the *Families-to-Persons* transformation, one of which is seen in Figure 6.9b. This rule is not guaranteed to apply when *Families* exist with *Members*, as they may not be placed in a *Country*.



(a) *FourMembers* contract for the *Families-to-Persons* transformation.



(b) The *copersons…Man* rule from the *Families-to-Persons* transformation.

Figure 6.9 – Contract and rule to explain the interaction of pruning with contract-proving.

The post-condition of the contract checks for the presence of *Men* and *Women* elements which are connected to the *Community* element with the *persons* relations.

As the *persons* association is a containment link in the meta-model, then the pruning operation will prune away any case where these rules have not symbolically executed and the associations do not exist.

That is, when not pruning path conditions will be created where the associations do not exist and the contract will fail. When pruning is enabled, all path conditions must contain the associations and the contract will succeed.

6.3.3.1 Algorithm and Complexity

Our discussion of the pruning algorithm is split into two steps. The first step is to gather information about the containment links in the output meta-model for the transformation. This information is used to examine all the rules in the transformation and determine which elements in the rules require containment links, and if any containment links cannot be found in the transformation. This first step is performed before the path condition generation process begins.

The second step is performed during the generation process. Each path condition is examined during path condition construction to determine which containment links need to be built. It must be determined if the missing containment links are present within the path condition or the remaining rules in the transformation. If the links are not to be built, or not built enough times, then the path condition is removed.

Examining the Meta-model and Rules. The first step in the pruning process is to examine the output meta-model to determine which pairs of nodes have a containment relationship,. This involves iterating through all the nodes and edges in the meta-model, therefore the complexity is $\mathcal{O}(|V_{meta-model}| + |E_{meta-model}|)$.

Next, two sets of containment links are created by examining each rule. The first link set is the type and count of containment links that a rule requires. This is obtained by iterating over all apply nodes in the rule not attached to a backward link, and summing the containment links defined in the meta-model for each element. Note that this summation will lead to a rule requiring multiple instances of a containment link if multiple nodes demand it. The second link set is the set of containment links which have already been built in the rule.

Therefore, this first step in the pruning process produces a mapping from a rule to a set of containment links the rule requires, and a set of containment links the rule builds. This information can then be used to determine which rules produce a given containment link.

For example, in the *Families-to-Persons* transformation the *Community* $\xrightarrow{persons}$ *Man* containment link is required by the rules *Father2Man* and *Son2Man*, while the link is built by the two *copersons...Man* rules.

As well, it must be determined if there exists a containment link in the meta-model that does not exist in the transformation. If so, then that containment link is not considered to be required by nodes. This handles the situations where the transformation does not address some parts of the meta-model, or the transformation has been sliced for a contract (Section 6.3.2).

Note that extra care must be taken with rule multiplicity while pruning to ensure that rules are set to be symbolically executed an appropriate number of times such that all rules have their dependencies satisfied. The dependency analysis discussed in Section 6.4.1 on page 238 assists with this determination.

As all rules must be iterated, and each element examined in each, the complexity of this step is $\mathcal{O}\big((|V_r| + |E_r|) \cdot |Rules|\big)$.

Examining the Path Conditions. This containment link information can then be used for the pruning operation during path condition construction.

In particular, the pruning process can be performed at the end of each layer during the construction process. The aim of this is to remove as many path conditions as possible, before they are considered for combination with the rules in the next layer.

At the end of each layer, the path conditions are examined to determine which rules are present. Then, the sets of required containment links are summed together, to form a set of required containment links for the path condition.

The next step is to ensure that each required containment link is either built already in the path condition, or could be built by a rule in a future transformation layer. This is handled by iterating through the built containment links for both the rules present in the path condition and the rules yet to be symbolically executed, and subtracting these built rules from the required set of containment links. If there yet remains a required containment link that cannot be satisfied, then the path condition will be invalid and it is discarded.

For example, assume there is a path condition where the rules *Father2Man* and *Son2Man* rules have symbolically executed. This path condition would require two $Community \xrightarrow{persons} Man$ links to have been, or will be built. If the two *copersons...Man* rules have not been symbolically executed on this path condition, or if their layer has already been examined so that they will not be symbolically executed, then the path condition is not valid and it is discarded.

6.3.3.2 Results

Table 6.6 on the following page presents the results of applying the pruning optimization on our case studies. For most of these experiments, the parallelization and slicing optimizations were not used. The exception is the $mbeddr_{46}$ case study experiment conducted on our supercomputer machine, which uses parallelization but not slicing.

Note that in most case studies, pruning did not have a large effect on the number of path conditions generated. As well, there is a performance impact on generation time due to needed to examine path conditions. It is only in the *Families-to-Persons* and the larger *mbeddr* transformations where a dramatic reduction in path conditions is seen, which reduces generation and proving time as well as memory usage.

The difference in efficacy between case studies may be due to the difference in output meta-model size, as reported in Table 7.2 on page 250. We are currently investigating in which cases the pruning operation is most beneficial.

It is also important to emphasize that this pruning approach may change the result of contract proof, as shown for the *Families-to-Persons* transformation

Table 6.6 – Time taken for case studies with and without pruning.

Transformation	**Parameters**	**Num. PCs**	**Build PCs Time (s.)**	**Contract Proof Time (s.)**	**Memory (MB)**
RSS-to-ATOM	No pruning	10	0.17	0.05	41
	Pruning	5	0.16	0.05	41
GM-to-AUTOSAR	No pruning	10	0.24	0.13	42
	Pruning	9	0.25	0.12	42
UML-to-Kiltera	No pruning	645	1.72	3.87	49
	Pruning	558	1.97	3.72	48
Families-to-Persons	No pruning	4916	16.54	42.31	80
	Pruning	255	2.99	2.35	43
$mbeddr_{22}$	No pruning	1012	5.75	1.79	97
	Pruning	1012	6.10	1.79	97
$mbeddr_{34}$	No pruning	28018	67.10	23.33	163
	Pruning	14910	49.34	12.25	151
$mbeddr_{46}$	No pruning	1211888	1847.19	285.07	3815
	Pruning	625486	1112.48	142.00	1949

contracts in Section 7.4. Therefore, it is important for a user to be aware of this (optional) pruning optimization, and the potential effects on contract satisfiability.

6.3.4 Conclusion

This section has presented three algorithmic optimizations: parallelization, slicing, and pruning. Table 6.7 on the following page summarizes these optimizations. Note that the speedup column only considers the desktop machine.

The use of the parallelization optimization is recommended in all cases, as this optimization reduces the time taken for contract proving and has few drawbacks. In contrast, the slicing and pruning optimizations can vary in their results. As mentioned in Section 6.3.2 on page 222, the slicing optimization depends on the

Table 6.7 – Summary of algorithmic optimizations.

Optim. Name	**Speedup**	**Advantages**	**Disadvantages**
Parallelization	2.6x - 3.6x	-Straightforward division of path conditions -Approaches optimal speedup	-Hardware arch. affects effectiveness -Queue theory should be investigated -Difficult to profile
Slicing	3.8x - 72.6x	-Large increase in prover performance -Easier to understand counter-examples	-Effectiveness depends on contract and transformation interaction -Non-trivial to combine with dependency analysis and pruning optimization
Pruning	0.9x - 14.2x	-Possible decrease in time and memory used -Focuses set of counter-examples	-Performance overhead to examine path conditions -Effectiveness depends on transformation meta-model -Can change result of contract proof

particular contract and transformation. Dependencies between the contract and the rules may make it that no rules are removed and no benefit is seen.

The pruning optimization has even larger tradeoffs. In particular, pruning may change the result of contract proof as invalid path conditions are removed and not examined if they satisfy the contract. As well, there is a performance penalty when employing pruning, as each path condition must be examined during path condition generation to determine its validity.

6.4 Checks and Analysis

The algorithms described in Section 4.4, and the SyVOLT tool described in Chapter 5 verify contracts against the path conditions created from transformations. However, it is inevitable that errors arise during the construction of the transformation and contracts though typos and omissions.

To ensure that the transformation/contract designer is alerted to unintentional errors, a number of static and dynamic checks are performed during the path condition generation process. These checks verify that DSLTrans transformations and contracts are correctly built, in that they are consistent with each other.

The checks and analysis steps detailed in this section arose organically out of the development of the contract verification tool, and have been quite successful in detecting errors during our experiments. In particular, a number of typos in element names meant that contracts could never be successfully proved for the initial version of some of our hand-built transformations. These typos were successfully detected by the checks detailed in this section, allowing us to correct them.

As well, once the MPS front-end to our tool (Section 5.2.3) was developed, we were able to import our transformations and contracts. The projectional editor nature of MPS means that transformations and contracts must conform to the meta-model at all times. This detected further errors, suggesting that the projectional editor approach does reduce errors and development time.

The first analysis we perform is to detect the dependencies of rules and contracts to ensure that all needed elements can be satisfied. Second, the path condition construction process raises an error if a rule is not symbolically executed. Finally, upon a result for a contract (success/failure), we present an analysis report on which rules caused the contract to succeed or fail on the path conditions. An example and complexity discussion will be presented for each of these analyses.

The analyses discussed in this section are relatively simple, and yet offer a groundwork for understanding the results when the contract prover process fails. In particular, the analyses catch not only the trivial typos inherent in manual design of transformations and contracts, but also provide insight on how the contract relies on the rules in the transformation.

6.4.1 Dependency Checking

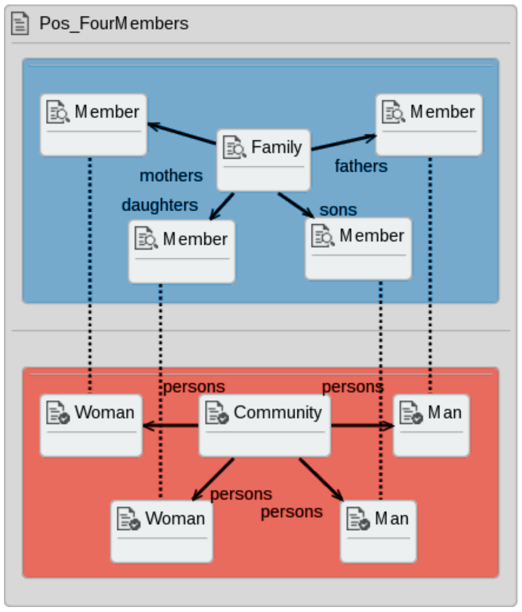
The first (and possibly most important) check we perform is on the dependencies of the transformation rules and contracts to make sure that all rules can execute, and that all elements in the contract could be produced.

This process begins in the same manner as is described in the slicing optimization (Section 6.3.2 on page 222). For each link and isolated node (a node not connected to any link) in the contract we check to see if it is present in some rule in the transformation. As well, we ensure that all backward links in the transformation can be satisfied by earlier rules.

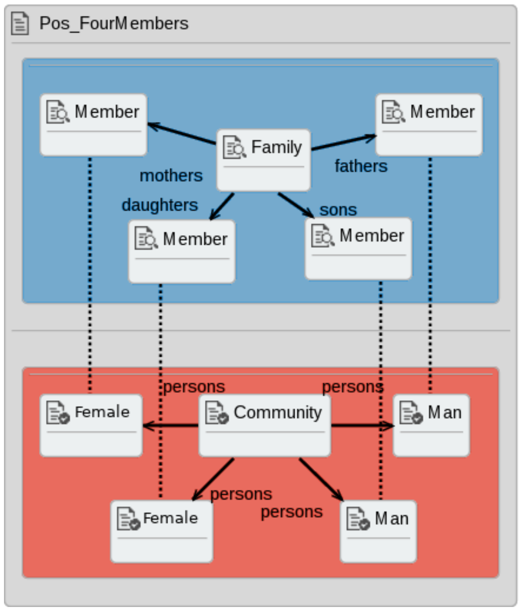
The multiplicity of elements is also checked during this step. For example, a rule may require two backward links between two types, but only one rule would produce these elements. The dependency checker will indicate that the producing rule may need to be symbolically executed multiple times (Section 3.2.5). This information can be passed to the path condition construction algorithm, ensuring that the correct path conditions are built.

This element-matching allows our analysis to check if there exists an element in a contract or a rule that cannot be satisfied in the transformation. In this case, even before the contract prover executes we know that there is an error with the contract or the transformation. When this occurs, the user is alerted to the precise missing element in the rule or contract so that it may be fixed.

Example. As an example of this analysis, we will make an intentional error in the *FourMembers* contract presented in Figure 6.10a. Instead of having *Woman* elements in the *Output* graph of this contract, let us instead have *Female* elements in the erroneous version in Figure 6.10b. This represents the case where the contract builder has unintentionally used an old meta-model or made a typo.



(a) *Woman* elements present in the *Output graph.*



(b) Replacing the *Woman* elements with non-existent *Female* elements.

Figure 6.10 – Original and erroneous *FourMembers* contracts.

When the contract prover is determining which rules the contract depends on, the output in Figure 6.11 on the next page will be produced.

In the section marked *a)*, the analysis indicates that the direct link from the *Community* element to the *Female* element cannot be found in the transformation. This therefore indicates that no path condition will exist where the contract's post-condition can be satisfied, and thus the contract will never hold.

The output marked *b)* states that there is also a backward link in the contract which cannot be satisfied by the transformation. That is, there is no rule that matches over a *Member* element and produces a *Female* element. To assist the user, the rules where these elements are presented are indicated in sections *c)* and *d)*.

The *c)* section mentions (a selection of) the rules that contain a *Member* element. Note that these rules may contain the element as a sub-class instead. For example, the *Father2Man* rule contains a *Parent* element which is a sub-class of the *Member*

Figure 6.11 – Dependency Analysis presented for the errorneous *FourMembers* contract.

```
a) Error: Elements in contract FourMembers direct link are missing:
Community - directLink_T - Female

b) Error: Backward link might be missing in contract FourMembers:
Member - trace - Female

c) Looking for meta-model element: ‘Member’
Rule Daughter2Woman contains meta-model element: ‘Member’ as ‘Child’
Rule Father2Man contains meta-model element: ‘Member’ as ‘Parent’
Rule Mother2Woman contains meta-model element: ‘Member’ as ‘Parent’
Rule Son2Man contains meta-model element: ‘Member’ as ‘Child’
...

d) Looking for meta-model element: ‘Female’
Error: Meta-model element ‘Female’ not found in any rule!
```

element. This allows the user to focus on the rules which are likely candidates for errors.

Finally, the section marked *d)* informs the user that no rule contains a *Female* element. This is marked as an error, as all elements of the contract must be present in the transformation. If not, the user must modify the contract or transformation such that the element exists.

Complexity. The complexity for the dependency checking analysis is exactly the same as the slicing analysis performed in Section 6.3.2. In fact, our implementation calculates the dependency information first, and then uses that for the slicing procedure.

Thus, the complexity of the dependency check is $\mathcal{O}(X \cdot |Rules|^2)$, where X is the complexity of the matching algorithm (Section 6.2).

6.4.2 Rule Application

The second analysis in our tool is performed during the path condition generation process itself, where it is detected if a rule was not symbolically executed. Note that the work of Selim [135] defined this check as a *rule reachability* contract.

A rule not symbolically executing may indicate an issue with the transformation where the required elements were not present, a multiplicity issue where a rule will not symbolically execute enough times, or an issue with the meta-model that has caused the pruner (Section 6.3.3) to remove all path conditions containing that particular rule. All cases indicates an issue with the transformation which should be reported to the user to investigate.

The current implementation for the rule reachability analysis involves checking all path conditions to determine which rules they abstract. This examination is performed at the end of a layer in path condition generation after the pruning step. If a rule appears in no path condition, then it either did not execute in that layer or it has been pruned away. An error is raised informing the user which rule did not execute.

The complexity of this operation is linear in terms of the number of path conditions [6]. Then, each rule is checked to make sure it was symbolically executed. Thus, the complexity is $\mathcal{O}(|PCs| + |Rules|)$.

A more efficient implementation may be to keep a mapping from rules to their path conditions during the construction process. Then this mapping would ensure that all rules were symbolically executed.

6.4.3 Contract Result Analysis

Finally, we present our analysis component which provides detailed information to the user at the end of contract verification to assist the user to understanding how a contract relates to the rules in the transformation.

6. Note that care must be taken to avoid de-serializing each path condition to examine elements, as this would be cost-prohibitive. Our current implementation is able to detect the rules involved from the name of the path condition.

Recall that at the end of contract proving (Section 4.4 and Section 5.3.5), we collect a set of path conditions where the contract holds, and a set where the contract does not hold. Let these be the *successful* and *failed* path condition sets.

The analysis component examines these sets and provides the following information:

— $rules_{success}$ - The rules common to all successful path conditions

— $rules_{fail}$ - 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 $rules_{success}$

— 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

   — The user can use this information and a visualization of the path condition to better understand the failure of the contract, as in Figure 6.14 on page 246

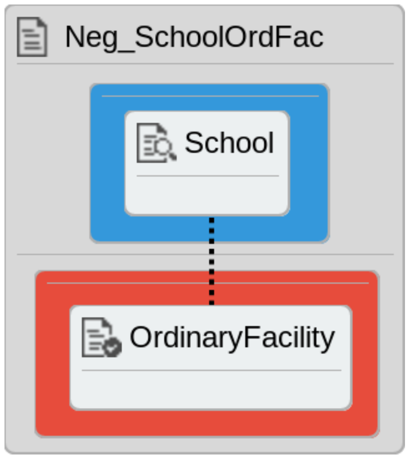
This comprehensive information allows the user to precisely determine how the contract is succeeding or failing, and have assurance that the result is as intended.

6.4.3.1 Example

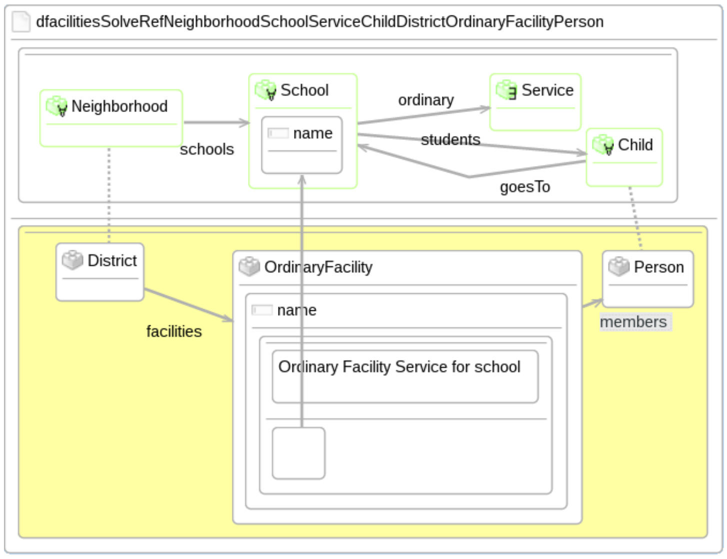
We will present the contract in Figure 6.12a as an example for the information we present when a contract has counter-example path conditions.

Note that for this contract we expect counter-examples to be found. This contract is from the *Families-to-Persons* transformation (Section 2.2), and represents the statement 'Every *School* in the input model will produce an *Ordinary Facility*'. The rule *dfacilities...OrdinaryFacilityPerson* which performs this production is displayed in Figure 6.12b.

However, in the *Families-to-Persons* transformation, there is also the *dfacilities...SpecialFacilityPerson* rule, which matches over a *School* element with a *special Service* to produce a *Special Facility* element. Thus, the contract is expected to fail on path conditions that include the *dfacilities...SpecialFacilityPerson* rule.



(a) An example contract for analysing success/failure results.


dfacilitiesSolveRefNeighborhoodSchoolServiceChildDistrictOrdinaryFacilityPerson
Neighborhood
School
name
Service
Child
ordinary
students
schools
goesTo
District
OrdinaryFacility
Person
name
facilities
members
Ordinary Facility Service for school

(b) Rule needed for contract success.

Figure 6.12 – Contract and rule examples for contract result analysis.

Figure 6.13 – Contract Result Analysis presented for the *SchoolOrdFac* contract.

```
a) Name:SchoolOrdFac
Num Succeeded Path Conditions: 6
Num Failed Path Conditions: 3

b) Explaining why contract fails: SchoolOrdFac
Good rules: (Rules in success set and not failure set)
dfacilities...OrdinaryFacilityPerson
Bad rules: (Rules common to all in failure set)
dfacilities...SpecialFacilityPerson

c) Contract requires elements from successful rules of type:
School
OrdinaryFacility

d) Examining failed pc:
Son2Man-Neighborhood2District-dfacilities...SpecialFacilityPerson

e) Elements of pattern that fail on this target:
Pattern could not find links of type:
OrdinaryFacility - trace_link - School
```

For the *SchoolOrdFac* contract, the contract prover indicates that there are six path conditions where this contract holds, and three path conditions where it does not. This output is seen in Figure 6.13 on the line marked *a)*. Following this is the contract result analysis for this case.

The contract result analysis indicates in the section marked *b)* that the *dfacilities...OrdinaryFacilityPerson* rule is common to all the successful path conditions. This is correct, as this rule must symbolically execute for the contract's pre- and post- condition to match. As well, the *dfacilities...SpecialFacilityPerson* rule is common to all of the failed path conditions. This is sensible as this rule produces path conditions where the *School* element in the contract pre-condition is found, but the *OrdinaryFacility* element is not. Note that the analysis will ignore rules that are both in the successful and failed path conditions to ignore common rules such as the *Son2Man* rule.

The last two analysis precisely identify *why* the contract fails on the path conditions. Based on the dependency analysis performed in Section 6.4.1, the contract prover has a record of which elements and links the contract depends on for each rule. In this example, the contract depends on the *School* and *OrdinaryFacility* elements produced by the *dfacilities...OrdinaryFacilityPerson* rule.

As seen on the line marked *d)*, the analysis picks a random path condition from the failed set of path conditions to examine further. A matching is performed with extra debugging information which indicates on the line marked *e)* that the *School* - *OrdinaryFacility* traceability link could not be found in the path condition. This path condition is represented in Figure 6.14 on the next page.

Visualization. Note that we intend to expand these analyses to better relay to the user why a contract succeeded or failed. In particular, we are interested in the visualization and debugging of the matching algorithm. Currently, path conditions can only be visualized with an extremely unintuitive representation.

For example, a path condition where the *SchoolOrdFac* contract fails is shown in Figure 6.14 represented as a collection of nodes and edges. This representation is inefficient to reason about and understand why a contract fails on this path condition, as the user must reconstruct the path condition in their heads as a combination of the match and apply components of three rules.

Another critique of this visualization is the inability to see the structure which would allow the contract to be satisfied. For example, the *Neg_SchoolOrdFac* requires a *School* element connected to a *Service* element through an *ordinary* link, not the *special* link shown in a pale yellow oval in the middle of Figure 6.14.

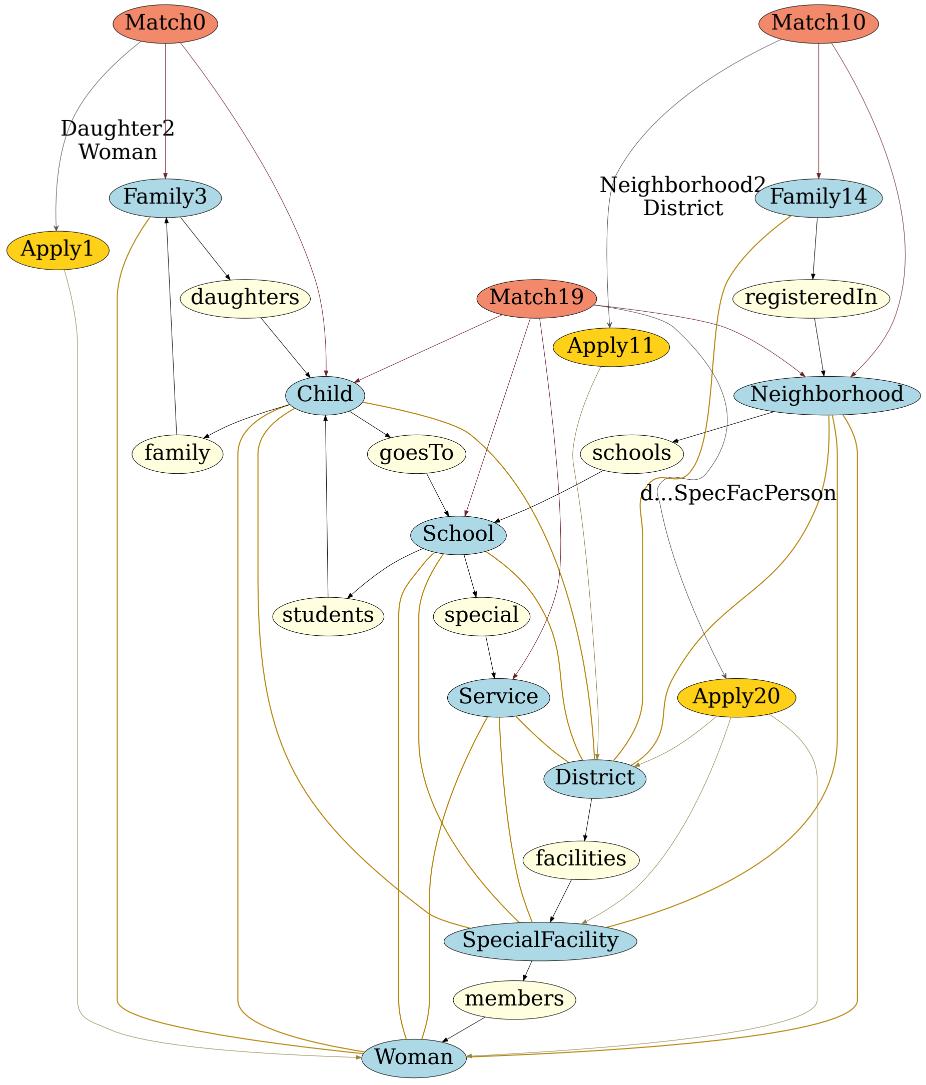


Figure 6.14 – An unintuitive visualization of a path condition that fails to satisfy the *SchoolOrdFac* contract.

In our collaborations with graph-based tool designers (such as the ModelVerse project [158] and the AToMPM team [146]), we aim to create and promote a reusable library for graph-matching visualization and debugging.

6.5 Conclusion

A primary focus of the development effort for SyVOLT has been on technical and algorithmic improvements over the symbolic execution approach defined by Barroca [19]. This significant effort was required because of the time and space requirements of producing large number of path conditions for each transformation.

In a contribution of this thesis, we have presented in Section 6.2 our *split morphism matching algorithm*. This algorithm is able to 'split' the nodes in the pattern graph over the target graph, which is a unique property that we employ in our path condition building and contract proving algorithms. Section 6.2 presents a description of the algorithm along with experimental results on a database of synthetic graphs.

A number of algorithmic techniques were also devised to decrease the time and space taken for operation of the prover. Section 6.3 discusses the standard technique of parallelization of the tool, as well as the domain-specific technique of slicing the transformation and pruning invalid path conditions.

As well, we have also provided in Section 6.4 some analyses that we perform during operation of the prover. These analyses ensure that a) the dependencies of the contract and rules in the transformation are met, b) that all rules symbolically execute during path condition construction, and c) the user is presented with additional information to understand the result of contract proving.

Chapter 7
Case Studies

This section will describe the case studies involved in our development of the contract prover. For each case study, we will present the origin and purpose of the transformation, a selection of the rules and contracts involved, as well as any relevant publications. These case studies are used in the preceding chapters of this thesis as the tests for our performance results, such as the parallelization and pruning optimizations in Section 6.3 on page 213.

Note that the contracts presented in this chapter have been selected to highlight interesting features of the contract language, and to provide an intuition of what information contracts can provide about a transformation. Future work will focus on the process of creating contracts such that a transformation is sufficiently verified.

7.0.1 Case Study Sizes

Table 7.1 and Table 7.2 present metrics for the size and complexity of the case study transformations and their respective meta-models. This provides an intuition about the range of transformations that our technique can verify contracts for. Note that the transformation may not cover the entire meta-model.

Table 7.1 on the next page contains the number of layers, rules, match elements, apply elements, associations, attributes, and backward links for each case study. Note that our case studies are ordered in increasing size and complexity.

Table 7.1 – Metrics for case study transformation size.

Transformation	**Num. Layers**	**Num. Rules**	**Graph**	**Num. Elements**	**Num. Assoc.**	**Num. Attrib.**	**Num. B. Links**
RSS-to-ATOM	7	7	Match	8	1	16	5
			Apply	14	7	14	
GM-to-AUTOSAR	8	9	Match	26	17	5	10
			Apply	20	10	9	
UML-to-Kiltera	8	17	Match	46	30	25	18
			Apply	91	74	58	
Families-to-Person	16	19	Match	46	39	14	24
			Apply	33	14	10	
mbeddr	6	46	Match	128	87	43	63
			Apply	147	102	24	

Table 7.2 on the following page displays information about the input and output meta-models for each transformation, and details the number of classes, associations, containment associations, and inheritance relationships present in each meta-model.

7.0.2 Contract Description

Contracts are detailed in this section by providing an informal statement of the intention of the contract, a graphical representation, and how many path conditions the contract is satisfied or not satisfied by. As well, the total time to prove each contract will be reported, including the end-to-end time for prover operation and only the use of the parallelization optimization. This total time thus represents the intended use of the prover by a transformer verifier.

As in Section 4.4, a contract's success on a path condition means that both the contract's pre-condition and post-condition (including backward links) matched onto the path condition, while failure means that the pre-condition matched, but the post-condition did not.

Note that this section will describe contracts that we expect to fail on some path conditions for the transformation. In these cases, a brief explanation will describe

Table 7.2 – Metrics for case study meta-model (MM) size.

Transformation	**Num. Classes**	**Num. Assoc.**	**Num. Containment**	**Num. Attrib.**	**Num. Inheri. Relations**
RSS-to-ATOM					
Input MM	9	16	9	45	0
Output MM	14	23	17	38	6
GM-to-AUTOSAR					
Input MM	6	10	5	5	0
Output MM	13	13	9	8	3
UML-to-Kiltera					
Input MM	42	51	20	6	41
Output MM	30	41	35	9	20
Families-to-Persons					
Input MM	11	21	11	3	7
Output MM	12	8	6	2	9
mbeddr					
Input MM	77	41	26	3	84
Output MM	66	30	24	2	71

the rule interactions that prevent the contract from holding. This information is provided by the contract analysis component of our contract prover, discussed in Section 6.4 on page 237.

These contracts therefore increase our confidence in the correctness of our transformation, as the prover will produce counter-example path conditions where the contract does not hold. As the path condition represents the application of a particular set of transformation rules, this allows the user to reason about the interaction between the rules, and determine whether the transformation is erroneous or not.

7.1 RSS2ATOM

As our simplest case study we present the *RSS2ATOM* transformation. This transformation has its origins in the ATL zoo [2], and was converted to DSLTrans by our higher-order transformation, as described in Section 5.2.1 on page 164.

The transformation is designed to transform models from the RSS language to the ATOM language. Both RSS [10] and ATOM [108] are structured markups for a web feed, where a user can subscribe to receive data such as blog posts or podcasts.

For example, the RSS meta-model contains elements such as *Channels* with a *title* and *language*, while the ATOM meta-model contains elements such as an *Entry* with multiple *Authors*.

7.1.1 Transformation

The *RSS2ATOM* transformation is presented in Figure 7.1. Note that our higher-order transformation (Section 5.2.1) produces DSLTrans transformations with a particular layout, where each layer contains one rule. As such, the rules can be grouped into two parts. The first part, consisting of the rules up to *Item2Entry*, matches elements in the input graph and creates elements and attributes in the output graph. The second part of the transformation detects previously-built elements using backward links, and creates associations in the output model.

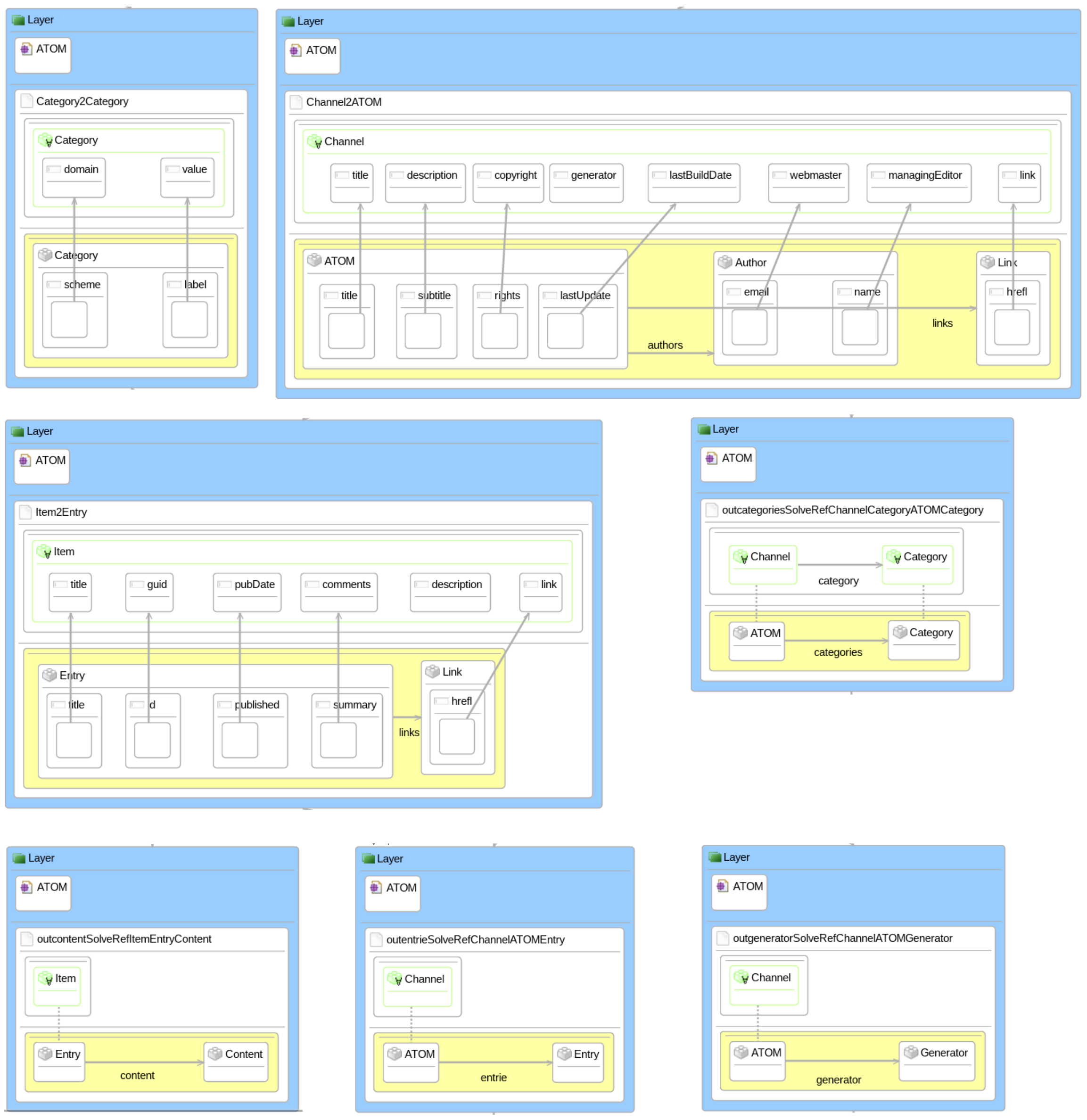


Figure 7.1 – The *RSS2ATOM* transformation.

7.1.2 Contracts

The contracts presented in this section were created by the author as simple examples for verification of the *RSS2ATOM* transformation.

Channel Production

Statement: A *Channel* in the input model should always produce an *ATOM* element connected to both an *Entry* and a *Generator* element in the output model.

Expected Result: Holds for all path conditions

Path Conditions Succeeded On: 6

Path Conditions Failed On: 0

Path Conditions Succeeded On w/ Pruning: 4

Path Conditions Failed On w/ Pruning: 0

Total Time to Prove: 2.04 seconds

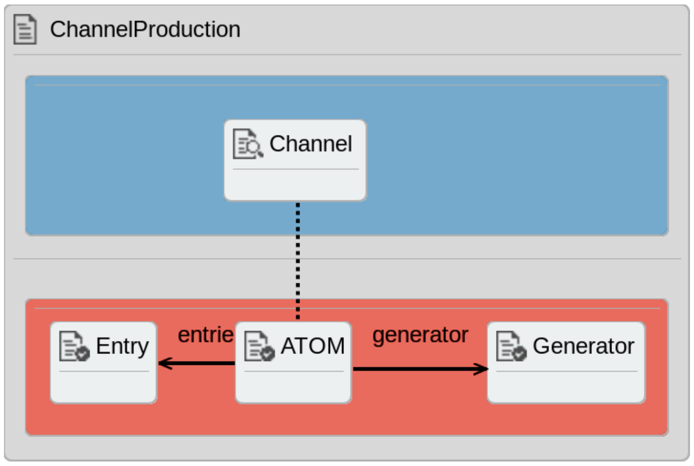


Figure 7.2 – *Channel Production* contract.

Explanation: The *Channel2ATOM* rule will produce the necessary *ATOM* element. Then, as both the last two rules in the transformation only match on an *ATOM* element produced by a *Channel* element, all three rules will necessarily execute together.

Entry Content

Statement: If an *Entry* element exists in the output model, then that *Entry* element should be connected to a *Content* element.

Expected Result: Does not hold for all path conditions

Path Conditions Succeeded On: 5

Path Conditions Failed On: 3

Path Conditions Succeeded On w/ Pruning: 2

Path Conditions Failed On w/ Pruning: 2

Smallest Successful Rule Set: {*Item2Entry*, *outcontentSolveRefItemEntryContent*}

Smallest Failed Rule Set: {*Channel2ATOM*, *outentrieSolveRefChannelATOMEntry*}

Total Time to Prove: 2.04 seconds

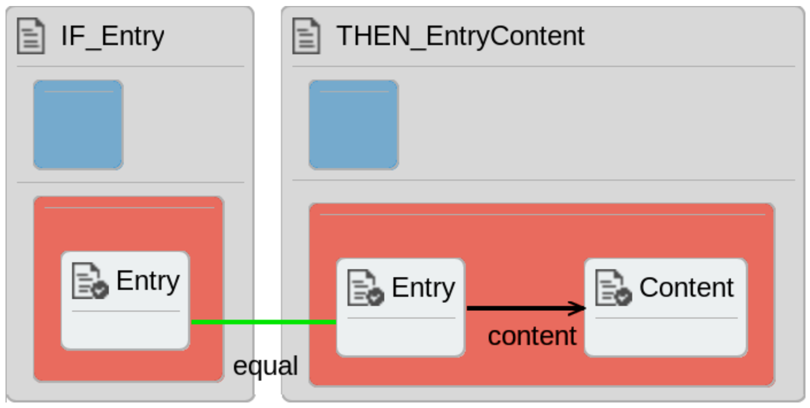


Figure 7.3 – *Entry Content* contract.

Explanation: The successful rule set produces an *Entry* connected to a *Content* element. In the failure rule set, an *Entry* element is produced that is connected only to an *ATOM* element.

Note that this contract uses the *If-Then* construct detailed in Section 4.6 on page 143. This means that the *if* contract must hold on the path condition for the *then* contract to be considered, and that both components must match for the contract to hold on the path condition. The *equals* line between the contracts the figure means that the same element must be matched over by both components of the *If-Then* construct.

7.2 GM-to-AUTOSAR Transformation

The following two transformations were developed in collaboration with colleagues at Queen's University. This collaboration led to a number of publications and a PhD thesis, showing how our technique is applicable to industrial case studies [137, 136].

One of the transformations produced during this collaboration was an industrial transformation termed *GM-to-AUTOSAR* [137]. The transformation operates on input models defined in a proprietary legacy metamodel used at General Motors (GM) for Vehicle Control Software development. The output metamodel is the automotive industry-standard AUTomotive Open System ARchitecture (AUTOSAR) [4]. Therefore, this transformation is intended for model-evolution purposes, specifically migrating the input models to the new standard for greater interoperability with tools.

As there are concerns with propriety information in the transformation, we refer the interested reader to the thesis of Selim [135] for a description of the transformation, metamodels, and the remainder of the contracts.

7.2.1 M6 Contract

This section presents the *M6* contract from the *GM-to-AUTOSAR* transformation. This contract was selected due to the subtle semantics implied by the use of negation in the contract.

M6

Statement: If an *System* element exists in the output model, then that *System* element should be connected to a *SystemMapping* element, and not (always) connected to two *SystemMapping* elements.

Expected Result: Holds for all path conditions

Path Conditions Succeeded On: 8

Path Conditions Failed On: 0

Path Conditions Succeeded On w/ Pruning: 8

Path Conditions Failed On w/ Pruning: 0

Total Time to Prove: 2.54 seconds

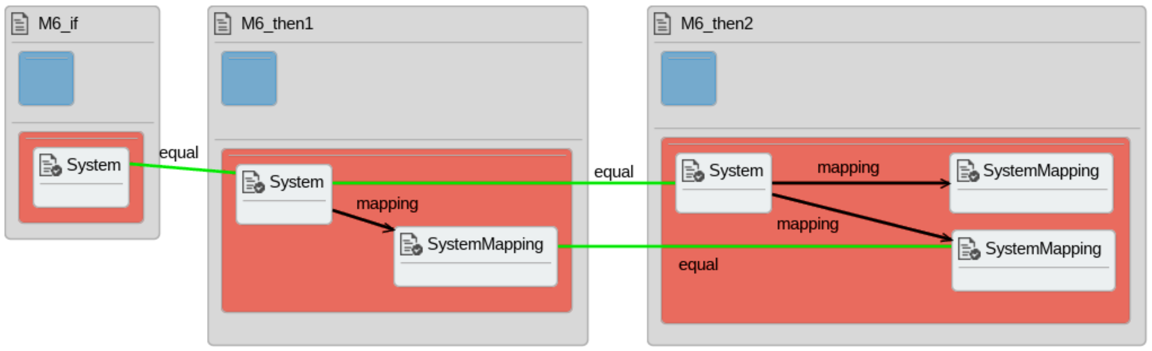


Figure 7.4 – *M6* contract.

The *M6* contract also displays the use of the propositional contract language described in Section 4.6. In this case, the contract is made up of three sub-contracts, as shown in Figure 7.4. The intention is that if the left contract matches, then the centre contract should match, and not the right contract. This contract therefore uses the *If-Then*, *And*, and *Not* constructs present in the contract language.

We also wish to bring attention to the *always* qualifier in the statement for the contract. As explained in Section 4.6 on page 143, the presence of the *Not* operator subtly changes the intention of the contract.

7.3 UML-to-Kiltera Transformation

As another case study, we examined a transformation for transforming a subset of UML-RT state machine diagrams into Kiltera, which is a language for timed, event-driven, mobile and distributed simulation [120].

The transformation was proposed in [117] and developed in [121]. This transformation was previously studied in our research to obtain insights into the contract-proving process [136], and was also a case study for earlier contract proving publications [112, 113].

Due to the size and complexity of the *UML-to-Kiltera* transformation, we refer the reader to the thesis of Selim [135] for full details, including the meta-models and majority of contracts for this case study.

7.3.1 Contracts

The *PP3* and *SS1* contracts have been selected for presentation as they demonstrate complicated behaviour. The *PP3* contract contains many input and output elements as well as many backward links and two attributes to match. The *SS1* contract is relatively simple but produces a non-trivial set of counter-examples.

PP3

Statement: A *composite State* element connected to *ExitPoint* and *Transition* elements should produce an *Inst* element connected to four *Name* elements, as well as a *Trigger* element with a *sh_in channel*.

Expected Result: Holds for all path conditions

Path Conditions Succeeded On: 108

Path Conditions Failed On: 0

Path Conditions Succeeded On w/ Pruning: 108

Path Conditions Failed On w/ Pruning: 0

Total Time to Prove: 6.82 seconds

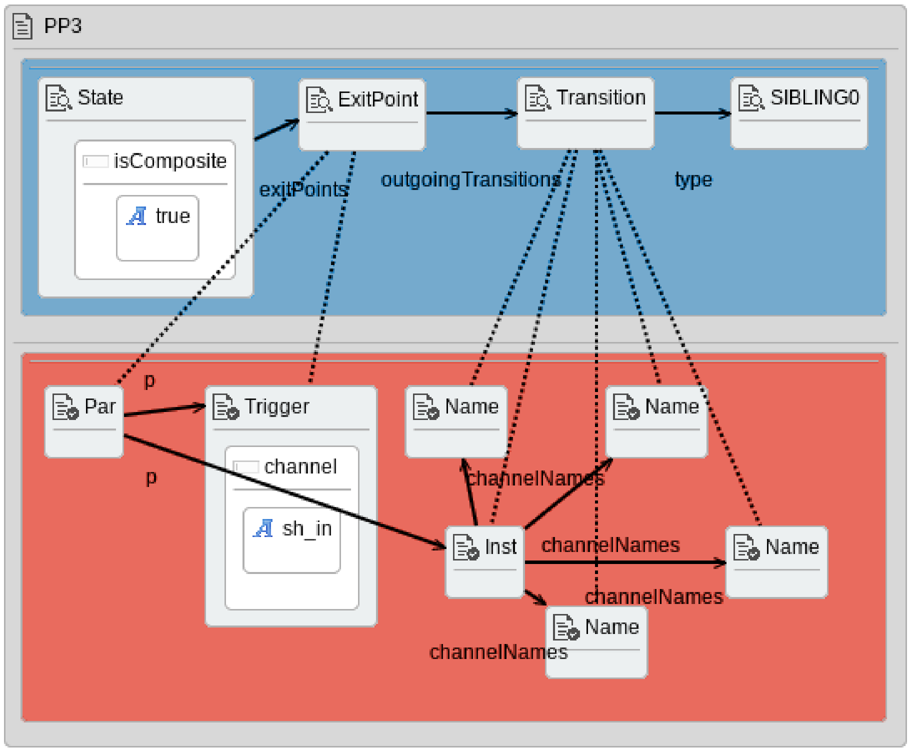


Figure 7.5 – *PP3* contract.

SS1

Statement: If an *Inst* element exists, there must also exist a *ProcDef*, where the *name* of the *ProcDef* is the same as the letter *S* added to the *name* of the *Inst* element.

Expected Result: Holds for all (non-trivial) path conditions

Path Conditions Succeeded On: 621

Path Conditions Failed On: 21

Path Conditions Succeeded On w/ Pruning: 549

Path Conditions Failed On w/ Pruning: 7

Total Time to Prove: 7.16 seconds

Smallest Successful Rule Set: {*State2ProcDef*, *BasicStateNoOutgoing2ProcDef*, *BasicState2ProcDef*, *CompositeState2ProcDef*, *State2HProcDef* }

Smallest Failed Rule Set: {*Transition2Inst*}, or {*Transition2QInstOut*}, or {*Transition2QInstSIBLING*}

Explanation: If only these three rules execute (in any combination), then an *Inst* element is produced but a *ProcDef* element is not.

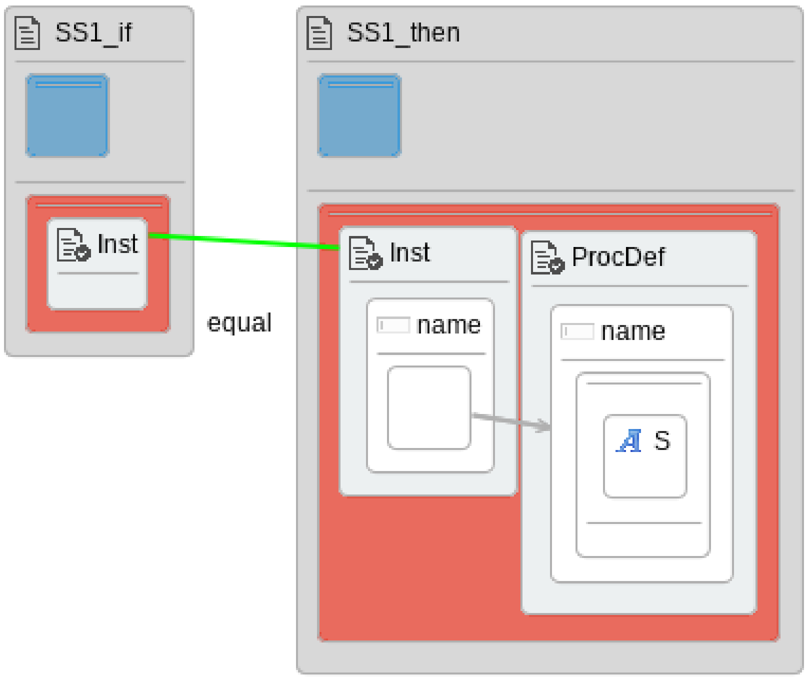


Figure 7.6 – *SS1* contract.

7.4 Families-to-Person Transformation

The main experiment for our contract prover development is the *Families-to-Persons* transformation, as described in Section 2.2.

This case study provides excellent motivation for the use of our non-isomorphic matching algorithm (Section 2.3.2.2 and Section 6.2). For example, multiple rules and contracts in the *Families-to-Persons* transformation contain the *Family* element. Our contract prover must be able to correctly perform non-isomorphic matching of the contract onto a path condition such that if the *Family* element exists in two rules in the path condition, the contract prover must resolve whether these elements match over separate elements in the input model, or over the same element. This example is presented in the section on contract verification in Section 4.4.2.3 on page 133.

As well, even though this is a synthetic (non-industrial) transformation, it is one of the larger transformations we have studied as shown in Table 7.2. As such, it provides an interesting case study for the implementation optimizations discussed in Section 6.3.

Figure 7.7 on the following page and Figure 7.8 on page 262 display the *Families-to-Persons* transformation, which contains nineteen rules each in their own layer. This structure is because this transformation was created automatically through a higher-order transformation, which converts an *Atlas Transformation Language* (ATL) [3] transformation into DSLTrans. This higher-order transformation is further discussed in Section 5.2.1.

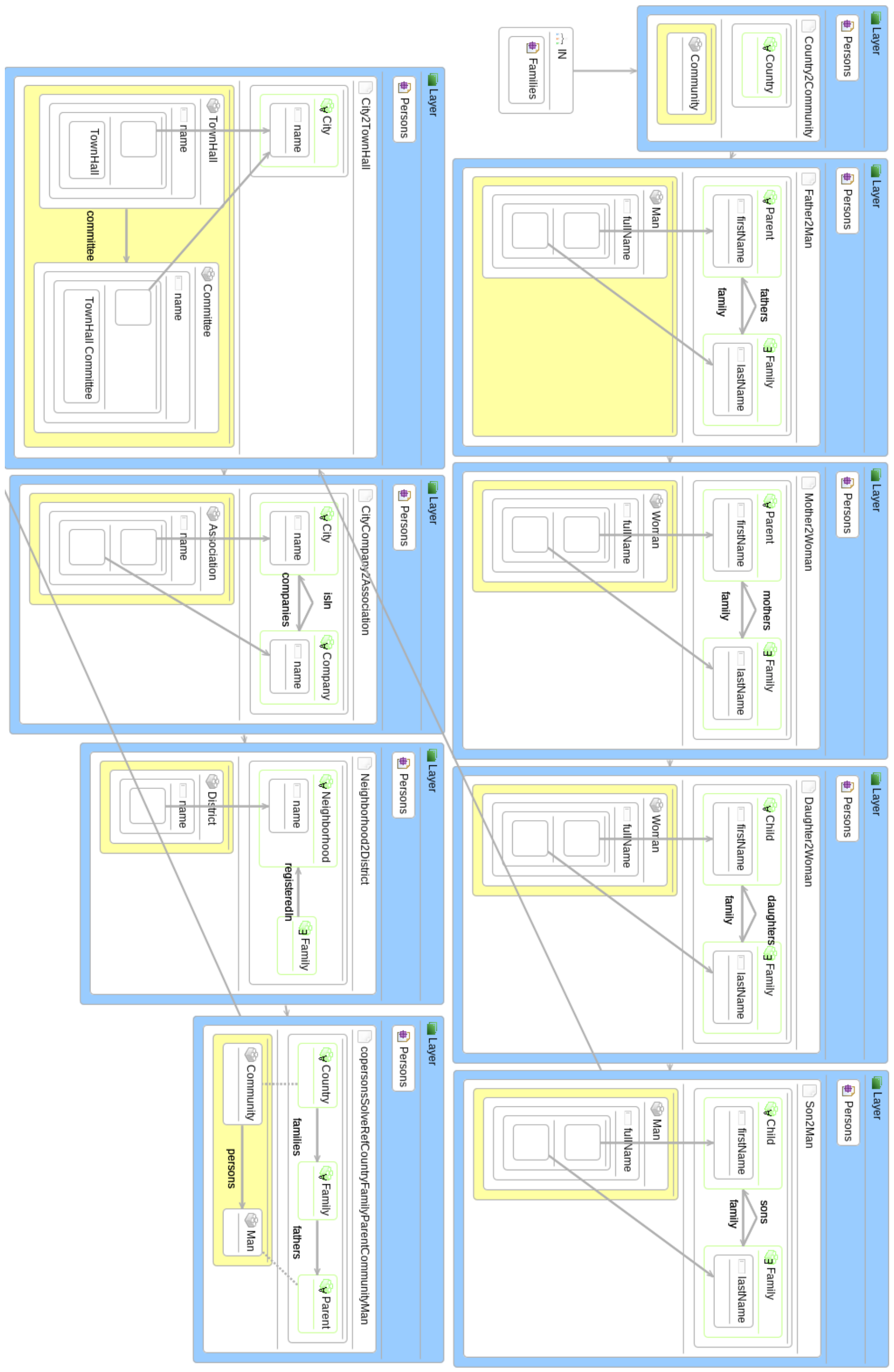


Figure 7.7 – The first nine rules of the *Families-to-Persons* transformation.

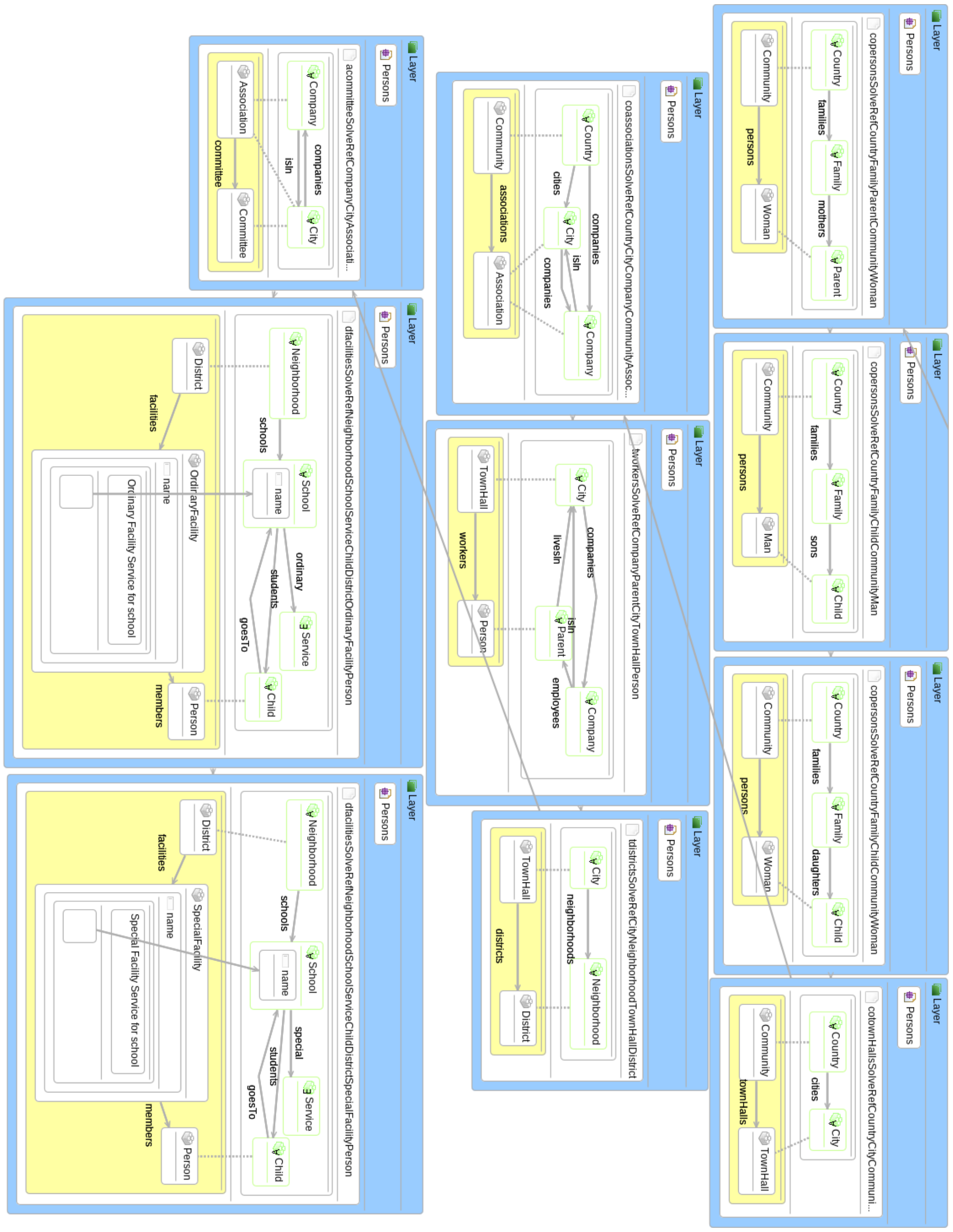

copersonsSolveRefCountryFamilyParentCommunityWoman
copersonsSolveRefCountryFamilyChildCommunityMan
copersonsSolveRefCountryFamilyChildCommunityWoman
cotownHallsSolveRefCountryCityCommuni...
coassociationsSolveRefCountryCityCompanyCommunityAssoc...
tworkersSolveRefCompanyParentCityTownHallPerson
tdistrictsSolveRefCityNeighborhoodTownHallDistrict
acommitteeSolveRefCompanyCityAssociati...
dfacilitiesSolveRefNeighborhoodSchoolServiceChildDistrictOrdinaryFacilityPerson
dfacilitiesSolveRefNeighborhoodSchoolServiceChildDistrictSpecialFacilityPerson
Layer
Persons
Ordinary Facility Service for school
Special Facility Service for school

Figure 7.8 – The last ten rules of the *Families-to-Persons* transformation.

7.4.1 Contracts

The majority of the contracts for the *Families-to-Persons* transformation are presented here. While the contracts were created synthetically, they aim to cover the meta-model, as well as provide examples for contracts that are not satisfied and examples of attribute usage.

Note that the following division of contracts into sections is primarily for readability, as they address different areas of the source and target meta-models.

7.4.1.1 Families-to-Persons Contracts

Pos_FourMembers

Statement: A *Family* with a *father*, *mother*, *son* and *daughter* should always produce two *Man* and two *Woman* elements connected to a *Community*.
Expected Result: Holds for all (valid) path conditions
Path Conditions Succeeded On: 113
Path Conditions Failed On: 493
Path Conditions Succeeded On w/ Pruning: 21
Path Conditions Failed On w/ Pruning: 0
Total Time to Prove: 14.99 seconds

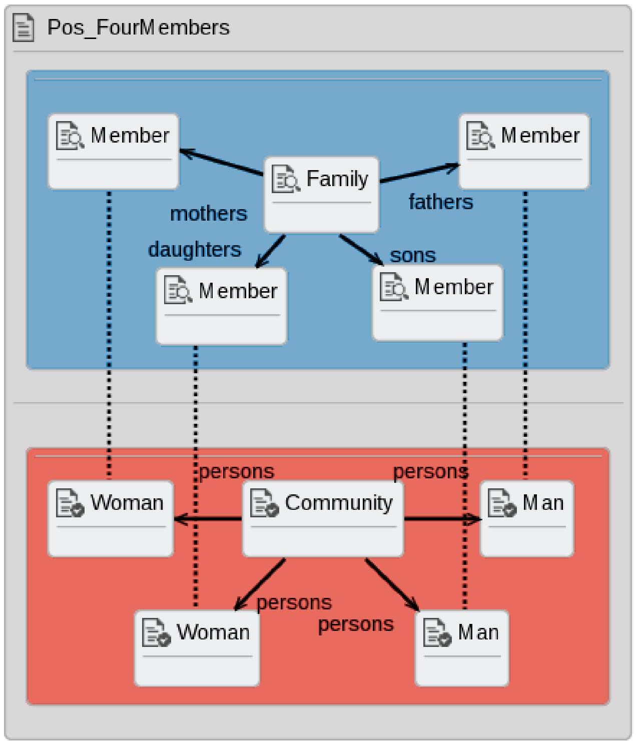


Figure 7.9 – *Pos_FourMembers* contract.

Smallest Successful Rule Set: {*Country2Community*, *Father2Man*, *Mother2Woman*, *Daughter2Woman*, *Son2Man*, *copersons...ParentCommunityMan*, *copersons...ParentCommunityWoman*, *copersons...ChildCommunityMan*, *copersons...ChildCommunityWoman*}

Smallest Failed Rule Set: {*Father2Man*, *Mother2Woman*, *Daughter2Woman*, *Son2Man*}

Explanation: Without pruning the invalid path conditions (as described in Section 6.3.3), path conditions are produced which do not properly connect to the *Community* element. Note the absence of the *copersons...* rules in the failure set as compared to the success set.

In Section 6.3.3 on page 229 we describe how these path conditions are invalid because of this missing *containment link*.

Pos_MotherFather

Statement: The full name of a produced *Person* is correctly created from the concatenation of the first name of the *Member* and the last name of his/her *Family*

Expected Result: Holds for all (valid) path conditions

Path Conditions Succeeded On: 716

Path Conditions Failed On: 1004

Path Conditions Succeeded On w/ Pruning: 72

Path Conditions Failed On w/ Pruning: 0

Total Time to Prove: 14.95 seconds

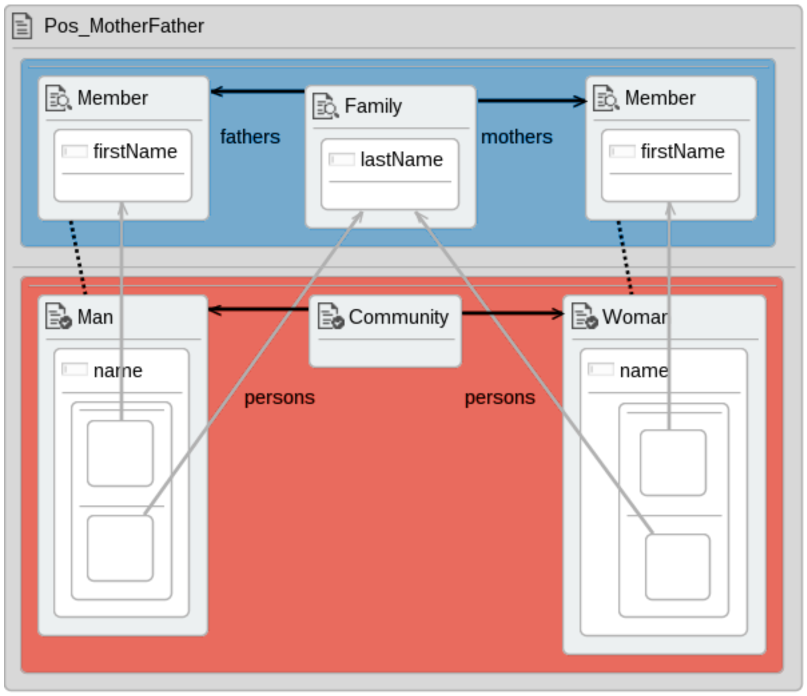


Figure 7.10 – *Pos_MotherFather* contract.

Smallest Successful Rule Set: {*Country2Community*, *Father2Man*, *Mother2Woman*, *copersons...ParentCommunityMan*, *copersons...ParentCommunityWoman*}

Smallest Failed Rule Set: {*Father2Man*, *Mother2Woman*}

Explanation: As with the *Four Members* contract, pruning removes those path conditions where the output elements are not connected to the *Community* element.

Neg_DaughterMother

Statement: A *Family* with a *mother* and a *daughter* will always produce a *Man*

Expected Result: Does not hold for all path conditions

Path Conditions Succeeded On: 1592

Path Conditions Failed On: 380

Path Conditions Succeeded On w/ Pruning: 63

Path Conditions Failed On w/ Pruning: 21

Total Time to Prove: 14.82 seconds

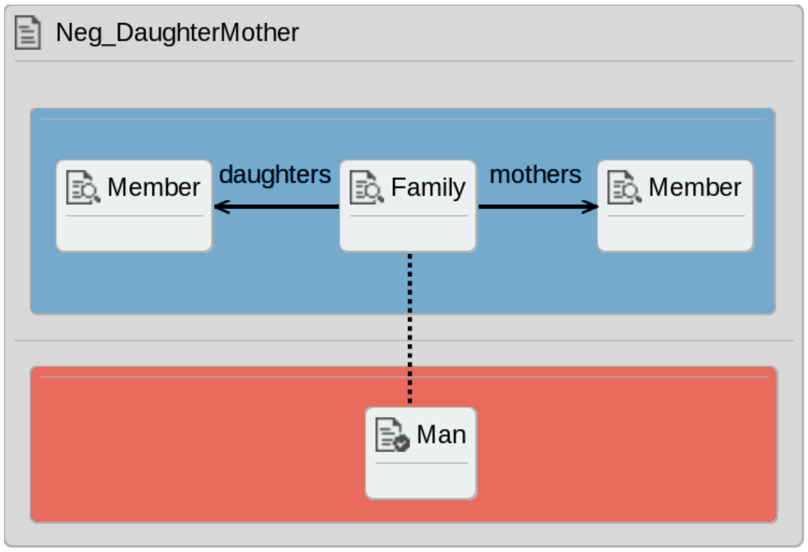


Figure 7.11 – *Neg_DaughterMother* contract.

Smallest Successful Rule Sets: {*Mother2Woman*, *Daughter2Woman*, *Son2Man*} or {*Father2Man*, *Mother2Woman*, *Daughter2Woman*}

Smallest Failed Rule Set: {*Daughter2Woman*, *Mother2Woman*}

Explanation: A *Man* element will not be produced from an all-female *Family*.

7.4.1.2 Location Contracts

Pos_AssocCity

Statement: A *Community* that contains a *City* with a *Company* should produce a *Community* with a *TownHall* and a *Committee*

Expected Result: Holds for all path conditions

Path Conditions Succeeded On: 1722

Path Conditions Failed On: 702

Path Conditions Succeeded On w/ Pruning: 170

Path Conditions Failed On w/ Pruning: 0

Total Time to Prove: 14.93 seconds

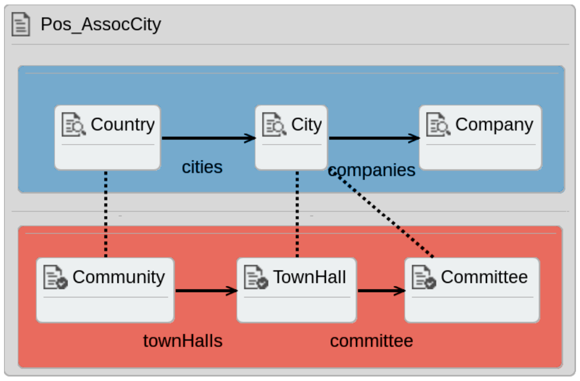


Figure 7.12 – *Pos_AssocCity* contract.

Smallest Successful Rule Sets: {*Country2Community*, *Mother2Woman*, *City2TownHall*, *cotownHalls...TownHall*, *tworkers...Person* } or { *Country2Community*, *CityCompany2Association*, *City2TownHall*, *acommittee...Committee*, *cotownHalls...TownHall*}

Smallest Failed Rule Set: {*Country2Community*, *CityCompany2Association*, *City2TownHall*, *acommittee...Committee*, *coassociations...Association*}

Explanation: Note that in the failure rule, the *cotownHalls...TownHall* rule is not executed. This only occurs when pruning is deactivated. In the meta-model, *Associations* are contained by *TownHalls* by the association built in this rule. Therefore, the path conditions where the contract fails only appears when pruning is not activated.

Pos_ChildSchool

Statement: If a *Child goesTo* a *School* that has a special *Service*, then a *SpecialFacility* has to be created that has the *Person* created from the *Child* as *members*

Expected Result: Holds for all path conditions

Path Conditions Succeeded On: 2024

Path Conditions Failed On: 0

Path Conditions Succeeded On w/ Pruning: 84

Path Conditions Failed On w/ Pruning: 0

Total Time to Prove: 14.65 seconds

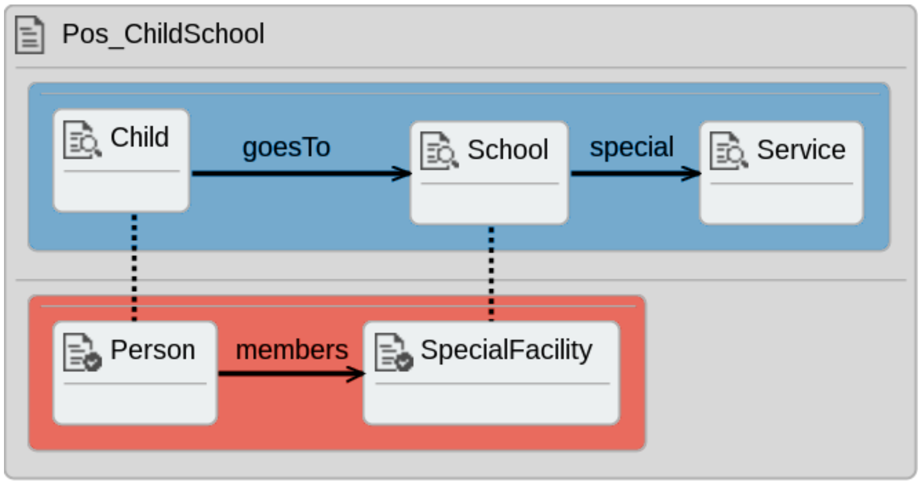


Figure 7.13 – *Pos_ChildSchool* contract.

Smallest Successful Rule Set: {*Country2Community*, *Daughter2Woman*, *Neighborhood2District*, *City2TownHall*, *copersons...Woman*, *cotownHalls...TownHall*, *tdistricts...District*, *dfacilities...SpecialFacilityPerson* }

Neg_SchoolOrdFac

Statement: An *OrdinaryFacility* should be created from each *School*

Expected Result: Does not hold for all path conditions

Path Conditions Succeeded On: 2024

Path Conditions Failed On: 1012

Path Conditions Succeeded On w/ Pruning: 84

Path Conditions Failed On w/ Pruning: 42

Total Time to Prove: 13.96 seconds

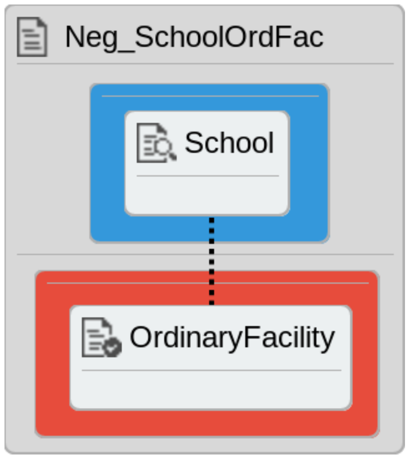


Figure 7.14 – *Neg_SchoolOrdFac* contract.

Smallest Successful Rule Set: {*Country2Community*, *Son2Man*, *Neighborhood2District*, *City2TownHall*, *copersons...Man*, *cotownHalls...TownHall*, *tdistricts...District*, *dfacilities...OrdinaryFacilityPerson*}

Smallest Failed Rule Set: {*Country2Community*, *Son2Man*, *Neighborhood2District*, *City2TownHall*, *copersons...Man*, *cotownHalls...TownHall*, *tdistricts...District*, *dfacilities...SpecialFacilityPerson* }

Smallest Successful Rule Set without Pruning: {*Son2Man*, *Neighborhood2District*, *dfacilities...OrdinaryFacilityPerson*}

Smallest Failed Rule Set without Pruning: { *Son2Man*, *Neighborhood2District*, *dfacilities...SpecialFacilityPerson* }

Explanation: A *School* will be transformed into a *SpecialFacility* if it provides a special *Service*. Also of note is that when pruning is deactivated, the path conditions indicated precisely specify the problematic rules.

7.5 mbeddr

The *mbeddr* transformation is a selection of rules from the open-source mbeddr project [9]. The aim of the mbeddr project is to provide high-level constructs for the incremental, modular, and domain-specific extension of C with a focus on embedded software development [169]. C code is mixed with mbeddr code inside the Meta Programming System editor [8] to define constructs such as interfaces and components, and then C code is generated using transformation rules. This approach has been shown to increase the quality of software by raising the level of abstraction [168].

This case study is currently under development in collaboration with itemis AG, fortiss GmbH, and Cláudio Gomes at the University of Antwerp.

7.5.1 Transformation

The *mbeddr* transformation is presented in Figure 7.15 on the next page. Note that we consider this transformation to be quite large, as it contains 46 rules of non-trivial complexity. As such, it is not feasible to explain every rule, and we present the figure only to provide a sense of scale and complexity.

Full Verification. Table 7.3 on page 272 shows the time and space characteristics when the *AssignmentInstance* contract is verified on the full $mbeddr_{46}$ transformation on the supercomputer. Note that the parallelization optimization was enabled to reduce time taken.

This shows that while the full *mbeddr* transformation can be verified, a powerful computer is required. Note that over 1.2 million path conditions were generated, which required 17 gigabytes of disk space. Our efforts will continue to bring these metrics down such that contract verification can occur on a desktop machine.

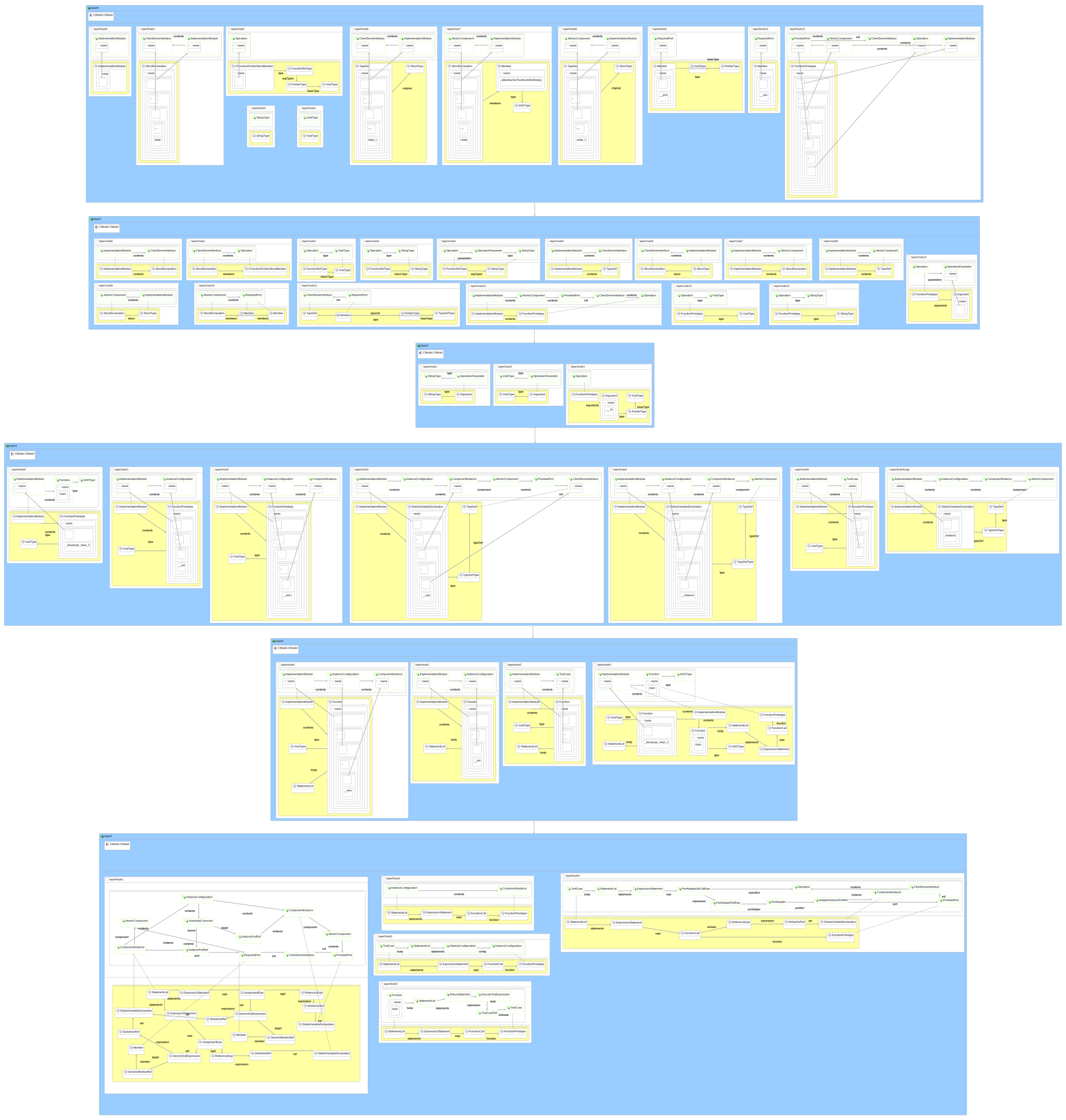


Figure 7.15 – The *mbeddr* transformation.

Table 7.3 – Performance when verifying the *AssignmentInstance* contract on the full *mbeddr*$_{46}$ transformation on the supercomputer.

Transformation	**Num. PCs**	**Build PCs Time (s.)**	**Contract Proof Time (s.)**	**Memory (MB.)**	**Disk Usage (MB.)**
Without Pruning	1211888	1847.19	285.07	3815	17949
With Pruning	625486	1112.48	142.00	1949	7452

7.5.2 Contract

Figure 7.16 shows the complicated *AssignmentInstance* contract for the *mbeddr* transformation.

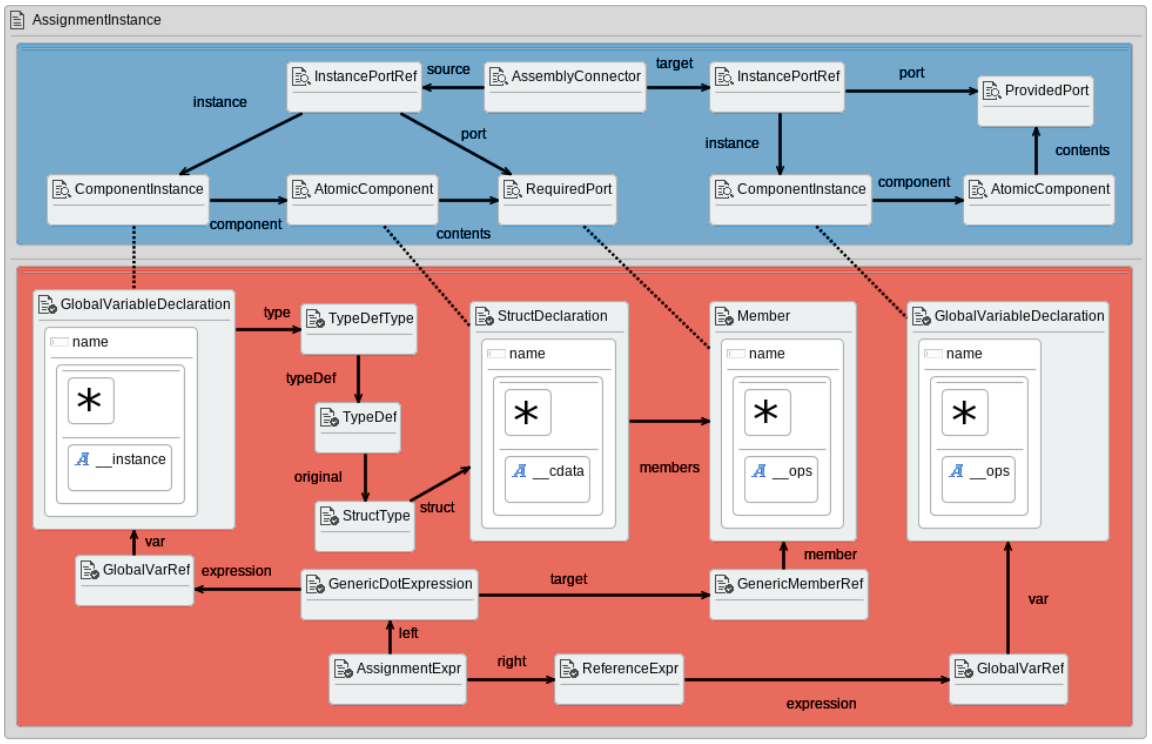


Figure 7.16 – *AssignmentInstance* contract.

Contract Intention. The *mbeddr* transformation defines a mapping between a high-level language and the C language, allowing a developer to abstract over concepts such as binding components to ports. This intention of this contract is to ensure that the transformation is correctly translating a binding between ports in the input model to a binding in the output model.

In the pre-condition of the contract, an *AssemblyConnector* element connects a source *RequiredPort* with the target *ProvidedPort*.

If these elements are present in the input model, then there must be an *AssignmentExpr* element produced. As well, the *left* association of the *AssignmentExpr* must be connected to a *GlobalVariableDeclaration*, a *StructDeclaration*, and a *Member*. Note that all three of these elements must have certain strings at the end of their name, as the Kleene star in Figure 7.16 indicates a *Wildcard element*.

The *right* association of the *AssignmentExpr* must also connect to another *GlobalVariableDeclaration*. produced from the *ComponentInstance* connected to the *ProvidedPort* in the input model.

Assignment Instance

Expected Result: Holds for all path conditions

Path Conditions Succeeded On: 2

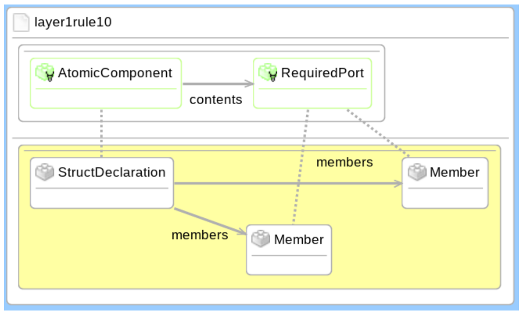
Path Conditions Failed On: 2

Explanation: The failure of this contract was unexpected when we were developing this case study. The contract analysis (Section 6.4) indicates that the critical rule which must execute for this contract to hold is *layer1rule10*, seen in Figure 7.17a on the next page. This rule connects the *StructDeclaration* element to the *Members* produced by the *RequiredPort*.

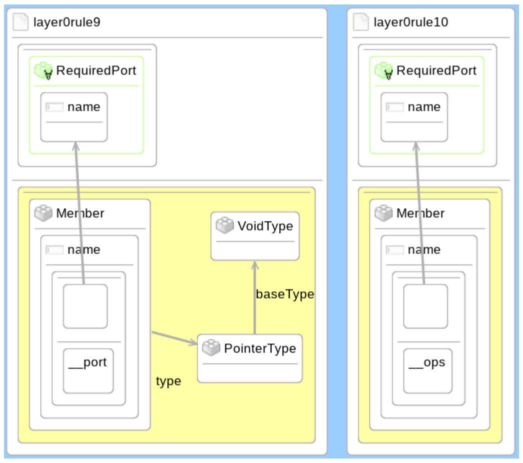
Another useful tool in the analysis of contracts is the prover's ability to determine the dependencies between rules. This feature is used in slicing, which reduces the transformation to only those rules needed for a particular contract (Section 6.3.2).

For this contract, the dependency analysis indicates that the *layer1rule10* rule depends on the *layer0rule7*, *layer0rule9*, and *layer0rule10* rules. The latter two rules are shown in Figure 7.17b. The critical observation here is that while the *layer0rule9*, and *layer0rule10* rules produce the necessary *Member* elements, the *layer1rule10* rule still requires the *contents* association between an *AtomicComponent* and the *RequiredPort*. Thus, there are possible transformation executions where the *layer1rule10* rule does not apply.

We are in the process of relating the insights gained through this contract debugging process back into the *mbeddr* transformation itself. In particular, the contract proof result highlights how critical the *layer1rule10* rule is. We are working with the transformation designer to determine if the transformation is at fault and needs to be changed to enforce that this rule always applies. Another solution is that the transformation is correct, and the contract must be changed such that the *layer1rule10* rule is assumed to always execute.



(a) The *layer1rule10* rule.



(b) Rules producing *Members* from *RequiredPorts*

Figure 7.17 – Rules critical to the proving of the *AssignmentInstance* contract.

Chapter 8
Related Work

This section will describe literature related to the topics presented in this thesis. First, comparisons between DSLTrans and other model transformation languages will be made. Second, works related to the proving of pre-condition/post-condition contracts will be presented. And finally, tools and algorithms related to various facets in our SyVOLT contract prover tool are highlighted.

8.1 DSLTrans

DSLTrans is a graph-based model transformation language. As such, it is similar to many other languages that also manipulate typed graphs in a rule-based way. This list includes AGG [147], VIATRA2 [18], ATL [74], and VTMS [92], which will be discussed throughout this section.

8.1.1 Graph Transformation Formalism

In this thesis, we have described the semantics of DSLTrans in the double-pushout approach, which is not the only formalism seen in the model transformation language literature [148].

For example, a conceptually-similar method to the double-pushout approach is the use of a *triple-graph grammar* (TGG) [134, 80]. A TGG relates structures in the source and target meta-models through a explicitly defined correspondence model. This approach allows the source and target model to be synchronized at all times, or to find the correspondences between a given input and output model. Many transformation languages use this underlying formalism [67] and a variety of tools exist to define and execute TGG transformations [61]. TGGs also support the use of attribute matching and rewriting in both the forward and backward direction [90, 68].

Note that this correspondence graph is similar to the traceability and backward links (Section 2.2.4) in DSLTrans. However, in TGGs the use of this correspondence graph allows bi-directional transformation, where the transformation can execute in one direction or the other. In DSLTrans, the transformation is restricted to execute only in the forward direction.

Future work will determine if the semantics of DSLTrans can be placed within the TGG approach. However, the double-pushout approach used in this work appears to be more applicable to the matching and rewriting steps performed in the contract verification portion of this research. In particular, it is unclear how the typed graph split morphism can be used in the TGG approach.

Models and transformations may be defined in a textual notion as well [45]. For example, the PETE transformation framework [132] can be used to create declarative, relational transformations in Prolog form.

The Maude language and framework provides a transformation language which is based on rewriting logic [42]. This logic can operate on state and non-deterministic concurrent computations by rewrite terms [128]. This logic may not be terminating or confluent, but Maude plug-ins provide analysis for these properties, as well as reachability and temporal logic properties. Riveria *et al.* also discuss the conversion of graph-based transformation specifications into Maude to benefit from Maude's analysis capabilities [129].

8.1.2 In-place versus Out-place

As described in Section 2.2, DSLTrans is an *out-place* and *non-deleting* language. This means that rules are permitted to only create elements in an output model, which must be different from the input model. This precludes the use of DSLTrans for simulation or refinement transformations [102], and as such we target translation transformations for our case studies (Chapter 7).

Other transformation languages do not have this restriction. For example, AGG is also based on algebraic graph transformations like DSLTrans, but allows for element deletion and model rewriting [147]. As well, Henshin [14] can execute *endogenous* (same input and output meta-models) and *in-place* transformations on EMF models.

Section 5.2.1 describes how ATL [74] can be used as an out-place translation language with similar semantics to DSLTrans. Our previous research of Oakes *et al.* [113] shows how a higher-order transformation can be made to convert declarative ATL transformations into an equivalent DSLTrans version. However, ATL can function in both the *in-place* and *out-place* modes, and any ATL transformation defined as an in-place (*refining*) transformation cannot be converted.

8.1.3 Control Flow

Some transformations languages and frameworks offer flexibility in the way that rules are scheduled. This provides the user with more expressiveness than the layering approach of DSLTrans, but then confluence and termination may not be guaranteed. For example, the T-Core transformation primitives described in Section 5.3.2 on page 180 allow for complex combinations to compose transformation languages.

The GreAT transformation language [77] is rule-based like DSLTrans, but has additional features such as a control flow language containing constructs for sequencing of rules, non-deterministic parallel firing of rules, hierarchy, recursion, and conditional branching. As well, the Fujaba tool [57] (which is based on the Progres graph transformation language) allows transitions in a control flow language with constructs such as repeating rules or applying a rule when another matches.

8.1.4 Confluence and Termination

Model transformation verification techniques may require that the transformation be *confluent* and/or *terminating* for proper analysis. However, these properties can be shown to be undecidable for graph-based transformation languages [118, 119].

As mentioned in Section 3.2, DSLTrans' transformation are *terminating* and *confluent* by construction. This is achieved through construction, as the out-place semantics and layer-based operation means that models can only monotonically grow, and that there are no looping constructs.

For other languages, different confluence criteria and analysis techniques are found in the literature [72, 87, 88]. For example, techniques for *critical-pair analysis* (Section 3.4.1) are found within the Henshin [33] and AGG tools [131] to identify those rules that conflict by removing or adding elements that other rules depend on. The work of Lambers *et al.* investigates the difficult notion of confluency when rules contain negative application graphs [89].

The problem of determining termination has led to a number of proposed criteria, as well as criteria analysis techniques, for transformations written in graph based transformation languages [47, 51, 166, 35].

An example of termination criteria is the creation of subsets of rules to examine termination dependencies, as present in the work of Bisztray and Heckel [31]. In that work, a graph is created to represent the interaction of rule negative application conditions with element creation and deletion. An acyclic dependency graph then demonstrates that the transformation will terminate.

The work of Biermann *et al.* [30, 29] focuses on formulating Eclipse Modelling Framework (EMF) model transformations [141] as typed graph transformations. The difficulty is that EMF models have containment relations such that there must be a path from a node to a root node. Therefore, Biermann *et al.* specify refactoring rules as algebraic graph transformations which obey the containment relation constraints. These refactoring rules are then shown to be confluent and terminating.

8.2 Model Transformation Verification

This section will discuss approaches for verification of model transformations. Properties of interest include termination, confluence, deadlocking or the guarantee that the transformation will preserve attributes of the original model.

However, the verification procedure is complicated by a number of factors, such as extremely large models or transformation, the wide variety of languages used to create transformations, and state-space explosion in determining all possible rule executions [123]. In this thesis, we tackle a number of these factors by employing the double-pushout approach, using our abstraction approach (Section 4.1), our match morphism which reduces the state-space (Section 2.3.2.2), and algorithmic optimizations to the contract prover (Section 6.3).

An approach to model transformation verification is to convert or compile the model transformation into another formalism for verification. This approach can leverage verification work performed in other domains. For example, the work of Gerking *et al.* [58] verifies reachability, safety, and liveness properties on a cyber-physical system model. This model is transformed into a network of timed automata, which is then verified using the UPPAAL tool suite [24]. The result of UPPAAL verification is combined with traceability links kept during the conversion from the original model to bring the counter-example back into the domain-specific language.

Another approach to model transformation verification involves semantic analysis of the transformation under study as well as the input and output languages. The work of Giese *et al.* [60] concerns relating the semantic equivalence of a transformation's arbitrary input model to a transformation and the programming language code produced as output. The verification result is obtained through conversion of the transformation to hundreds of lines of proof code, which is fed as input to a theorem prover.

In Barroca *et al.* [20], operational semantics are provided for the input and output languages using SOS. Then a DSLTrans transformation is provided which maps the input language constructs to the output language constructs. This approach is able to check that the semantics of the transformation are preserved and provide counter-examples if they are not.

8.2.1 State-space Analysis

An approach for model transformation verification is to represent the state-space of the transformation. This state-space can then be checked for properties of interest.

For example, the work of Varró *et al.* [165] explores the use of planner algorithms for model transformations. Planner algorithms are a technique from artificial intelligence research where an initial and goal state are described, and then the system tries to find a path of rules to traverse between them. In that work, the planner algorithm is used to prove that the paths of transformation rules conforms to a graph grammar.

A later paper by Varró focuses in generating a transition system for a transformation given the meta-model, the rules in the transformation, and a valid instance model [163].

The state space is then built by matching the rules onto the instance model in different combinations. The resulting transition system is then checked for a safety property and deadlock.

A similar transition system is also seen in work by König and Kozioura, where transformations are approximated by *Petri graphs* which contain model nodes with an overlaid Petri net [83].

The work of Becker *et al.* [23] proposes a technique for checking a dynamic system for safety properties by creating the state-space backwards from undesirable unsafe states. Counter-examples are then found by searching for invariants encoded in the rules of the transformation.

The GROOVE tool is similar to our research, as it can specify, play, and analyse graph transformations [59]. GROOVE also operates on a start instance graph, and then builds the state-space of rule application as graphs. For example, Rensink *et al.* [126] check for safety and reachability properties that are expressed as constraints over graphs by employing this state-space technique.

However, as with any state-space exploration problem, difficulties arise in avoiding a state-space explosion. A number of GROOVE studies [125, 22, 127] deal with this problem by introducing abstractions over the state-space. For example, [125] introduces the idea of *shape graphs*, which abstractly represent portions of the graph. Rule application may therefore effect only this abstracted portion of the graph, leaving the state unchanged.

This abstraction over the structure of the graph has parallels to our representation of path conditions as combinations of rules. In our work, we leave the issue of which rule elements overlap abstracted (Section 4.1), but we retain the full structure of each rule. As well, our abstraction relation defines a lower bound on rule application.

8.2.2 Contract Proving

The notion of contracts is to provide guarantees on the system under study. As described in Section 4.6, contracts can be divided into various classifications. The most common contract seen in the literature is the *pattern contract* type, or as termed in the classification of [26], a *syntactic* contract. Such contracts check whether certain elements in the input model will produce certain elements in the output model.

As an example of verification of semantic contract checking, the work of Dragomir *et al.* [49] defines *behavioural* contracts on SysML models. These models are transformed to the timed input/output automata formalism where the verification takes place.

The work of Akehurst and Kent [12] defines structural relations between the meta-models of the abstract syntax, concrete syntax and semantics domain of a fragment of UML. These triple-graph-grammar-like relations also encode OCL constraints [170] between the elements in the meta-model, which can be seen as an early (yet perhaps more powerful) version of the contracts we address in our work.

OCL constraints can also be used for the verification of model transformations themselves, such as in the work of Cariou *et al.* [39]. Their work defines a contract as constraints on the source model of a transformation, constraints on the target model. and constraints on 'element evolution'. These latter checks are defined as mappings between source model elements (classes, attributes), and target model elements. The technique therefore requires an input model and output model to the transformation, and the transformation must be endogenous.

Related OCL constraint work looks at the effect of transformation rules on the constraint. For example, the work of Clariso *et al.* [41] takes as input an OCL constraint and a transformation rule, and produces a constraint that holds before the application of the rule if and only if the original OCL constraint holds afterwards.

Narayanan and Karsai [107] propose verifying model transformations by writing structural correspondence rules between source and target meta-model elements (with optional attribute constraints). Similar to our contracts, these correspondence rules are written as pre-condition/ post-condition structures. The authors specify that during the execution of the transformation that *cross-links* (essentially our traceability links) are kept between input elements and the output elements that are produced by the input elements. Then, for one pair of input and output models, the correspondence rules are checked against the cross-links. Thus the correspondence rules prove that output elements have been successfully created. Note that while our work shares elements of the work of Narayanan and Karsai, our technique focuses on proving the structural contracts across all executions of the transformation.

The verification approach of Asztalos *et al.* [15] is based on the creation of an assertion language that allows making structural statements about models at a given point of transformation execution, as well as statements about the transformation steps themselves. These model transformation assertions can express a) properties of the models under transformation – “There are no elements of type T in the model”, b) properties of the rules – “Rule r terminates”, or c) the modification that is performed in the rules – “Rule r deletes all elements of type T from the model”.

The assertion framework also defines a number of *deduction rules*, which are formally defined in the double-pushout approach. Therefore, the framework attempts to derive the initial assertions in order to verify the properties. The deduction rule to be applied is selected based on the assertion at a particular step in the transformation. The framework presented is able to extract initial assertions from the transformation and prove properties automatically. However, the authors note that this derivation may take many derivation steps or even enter an infinite loop. Therefore a more specific reasoning system may be needed.

Another technique of Asztalos *et al.* [92] applies to transformations written in the VTMS transformation language. Note that this approach only requires the definition of the transformation and the meta-models of the source and target languages. No concrete input models are used for this verification, meaning that the result will hold for all possible input models, as with our approach.

Guerra *et al.* [69] 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 then tests a transformation by executing it on a number of these input models, and verifying them using the OCL expressions.

The approach 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 [66].

In an approach by Büttner *et al.*, an ATL transformation is mapped to a set of first-order logic statements [37, 38]. The set of ATL handled is restricted in order to only include terminating and confluent constructs, in an interesting parallel to our research which also requires the reduction of expressiveness. Then, a Satisfiability Modulo Theories (SMT) theorem prover can determine if all constraints can hold on the target models. This is performed through the use of a model finder which will search for the negation of a given OCL constraint that should hold. Note that the Alloy tool used performs bounded verification, and as such does not guarantee that a counter-example would be found if it exists. In contrast, our work will produce all counter-example path conditions for the contract.

In work by Gogolla and Vallecillo [63, 154], 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 meta-models, 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 meta-model, 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.

There are several other approaches supporting the testing of model transformations based on different kind of contracts such as model fragments [105], graph patterns [17], Triple Graph Grammars (TGGs) [172], dedicated testing languages [82, 56], and even a combination of these mentioned approaches [62].

8.3 SyVOLT Algorithms

This section will discuss work related to the algorithmic optimizations presented for our SyVOLT contract prover tool in Section 6.3. As well, we discuss works related to analysing and providing feedback for the transformation and its execution.

Note that the parallelization optimization we have performed on our prover (Section 6.3.1) is not unique to the transformation verification domain, and we refer the interested reader to works such as Kumar *et al.* [86] for a detailed examination.

Slicing. The slicing work on Section 6.3.2 shares similar characteristics to operations performed in other fields, such as compiler slicing of programs [149]. In that work, instructions in a program are selected as they relate to a property of interest.

Within the transformation verification literature, other approaches can be seen to be similar to our work. For example, the work of Burgueneo *et al.* [36] compares the elements in *Tracts* (discussed in Section 8.2.2) to the rules in the transformation. The intention is to suggest to the user which rules are causing the Tract to fail.

However, this use of slicing differs from ours in two fundamental ways. First, the approach of Burgueneo *et al.* focuses on guiding the user towards the problematic rules. In our implementation, this analysis is performed by the algorithms detailed in Section 6.4, where we report the interaction of rules that causes the contract to fail. Our slicing approach is solely a performance optimization to reduce the number of path conditions that must be created.

The second difference is that our sliced transformation absolutely must contain all rules that could change the result of a contract being satisfied or not by a transformation. We cannot afford a false result in our verification as is allowed in Burgueneo *et al.*, and thus our set of rules must be conservatively built.

The thesis of Ujhelyi [152] presents another expression of the slicing idea. In that work, slices are based on variables present in the declarative transformation specification, and the resulting slice is a combination of statements in the transformation, as well as individual model elements that are involved in the statements. Therefore, this is a *dynamic slice* in that the transformation is sliced for a particular model. In contrast, our slicing technique is not model-dependent, but is instead contract-dependent.

Pruning. The work of Biermann *et al.* [30] is related to that of our pruning approach defined in Section 6.3.3, in that both approaches rely on the presence of containment links in the meta-model of a model. Biermann *et al.* focus on defining and validating that refactoring rules for EMF models are consistent, in that containment links are respected at all times. In our research, these containment links are used to define invalid output path conditions that can be removed from the path condition generation process.

Note that *pruning* can also refer to other techniques in the literature, such as extracting the relevant portion of a meta-model [139].

8.3.1 Contract/Transformation Analysis

The analyses in Section 6.4 assist the user in generating a path condition set such that contracts can be proved. This is done by ensuring that dependencies are met, all rule symbolically execute, and providing further analysis for the contract result. These analyses share qualities with that found in the literature.

The thesis of Amstel [155] tackles the issue of evaluating the *quality* of a model transformation. User studies are performed to rate transformations based on qualities such as re-usability, completeness, and conciseness. The numeric ratings of these qualitative facets of the transformation are then correlated with quantitative metrics, such as the number of rules or variables in the transformation. This provides a guide for the transformation designer to build intuitive transformations. As well, the thesis details methods to visualize how the transformation covers the meta-model. While our approach is currently focused on how the contract result relates to the current transformation, we are examining how the prover can provide exact advice on increasing the quality of the transformation.

Also related to understanding the transformation is the ability to visualize and debug the execution of the transformation. As seen in Figure 6.14 on page 246, the SyVOLT tool currently produces a very unintuitive graphic representation of path conditions, which makes it difficult to understand how rules and contracts interact with that path condition. A better approach is seen in the thesis of Ujhelyi [152], which provides derivation rules to operate over the underlying model and produce a visualization. This reusable framework allows for the decoupling of the model and its representation.

Another form of debugging is to visualize the execution of the transformation, both before and after the application of rules as well as the actual matching and rewriting steps of each rule [101, 100]. In particular, during development of SyVOLT we found that we had two key questions regarding matching of the pattern graphs, *why is this not matching, when it should?* and *why is this matching, when it shouldn't?* We note that these debugging questions are relatively simple, and yet the igraph and T-Core libraries used lack sufficiently powerful debugging mechanisms.

In the literature, there has been recent success in explicitly modelling these steps in both the simulation domain [157] and for model transformations themselves [75].

Chapter 9
Conclusion and Future Work

The practice of model-driven engineering is today recognized as an essential part of industrial software development. As models become first-order artefacts for simulation, code generation, and knowledge representation, it is imperative to be able to reason about and modify these models in a structured way.

In this thesis, we have focused on reasoning about model transformations. In particular, our research is on the verification of translation model transformations. These model transformations take as input models which conform to one model language and produce an output model which conforms to another language. As seen in our industrial case study in Section 7.2, these translations can ensure that models are kept up-to-date in the industry standard language.

The reasoning that our verification research performs is based on structural contracts, which relate the elements in the input model to elements in the output model. Our technique builds a state-space for the transformation through the use of symbolic execution, and determines whether the contract is satisfied or not on each state in the state-space. If the contract is not satisfied, then a counter-example is produced for the user which represents a combination of rules which does not satisfy the contract. Thus, the user better understands the transformation through the production of these counter-examples.

A critical issue with state-space exploration techniques is the reduction of the state-space. In our research, we require that the transformation language be both terminating, confluent, and without deletion of elements. In our research, the DSLTrans transformation language fits these criteria.

9.1 Research Questions

The following sections will discuss the research questions posed in Chapter 1, our solutions to each question, and a brief discussion of future work.

9.1.1 Formalization of DSLTrans

Chapter 3 discusses our answer to the first research question: *How can the semantics of a model transformation language with reduced expressiveness (DSLTrans) be precisely formalized and brought to the state-of-the-art?*

Our research depends on the use of a model transformation language with restricted expressiveness and precise semantics. Previous work on DSLTrans demonstrated that this language provided the termination, confluence, and non-deletion properties we required for our path condition building and contract proving approaches. However, the precise semantics of all DSLTrans constructs were not given, and where given were not placed in the double-pushout formalization which is common in the model transformation literature.

Therefore, Chapter 3 defines the precise semantics of all DSLTrans constructs within the double-pushout approach. This approach centres on the idea of morphisms to represent the matching and rewriting steps of the rules in the transformation. Therefore, we have provided the matcher and rewriter structures for DSLTrans rules. These structures are employed in the creation of mass productions, which allow for the application of multiple rules at once onto a target graph. This enforces the property in DSLTrans where a rule is forbidden from matching on elements produced by another rule.

This thesis also makes the contribution of defining semantics for other constructs in DSLTrans. In particular, we discussed the presence of negative elements in DSLTrans rules. These elements prevent the application of rules through an extraction and matching procedure. As well, we have provided precise semantics for indirect links, which match over a transitive closure of the target graph, and *Exists* elements, which have restrictions on how they match over the target graph.

This formalization of the language is primarily important to enable us to apply the contract proving approach to DSLTrans. In particular, we require the specification of negative elements and indirect links to fully represent these in our symbolic execution approach. As well, the placing of DSLTrans in the double-pushout approach aligns us with other model transformation languages, such that we can explore related tools and approaches. This also increases the confidence of users of the transformation language, including our industrial partners.

9.1.2 Formalization of the Contract Proving Approach

In Chapter 4 we have presented a symbolic execution technique for the DSLTrans language, answering two of our research questions.

9.1.2.1 Abstraction Relation and Path Condition Creation

The first research question addressed was *How can the infinite execution possibilities of a transformation be represented in an explicit finite set?*

Our solution to this question is based upon the definition of an abstraction relation in Chapter 4.4. This relation allows us to create path conditions, which symbolically represent the application of transformation rules. These path conditions abstract input-output models, which are pairs of an input model to the transformation and the associated output model. The elements present in these input-output models are represented in the associated path condition, allowing us to determine how rule application creates elements in the input and output models. This abstraction relation is presented along with a list of examples which demonstrate how the conditions of the abstraction relation are necessary.

As a contribution of this thesis, we have discussed how our abstraction relation technique can represent multiple applications of a rule. This allows us to build path conditions which represent these multiple applications and therefore extend the range of contracts which we can prove. We also discussed the consequences of this multiplicity representation for path condition construction and contract proving.

Following the definition of the semantics for all DSLTrans constructs in Chapter 3, we now extend our path condition building technique to all DSLTrans transformations. Therefore, negative elements, indirect links, and *Exists* elements are now addressed within our technique.

9.1.2.2 Contract Proving Approach

Chapter 4 also addresses a two-part research question: a) *How can structural pre-condition/ post-condition contracts be proven to be satisfied or non-satisfied on these representations?* b) *When a contract is not satisfied, how do the counter-examples produced relate to the transformation?*

Part a) is discussed in Section 4.4, where we have presented our contract verification technique. The contracts of interest are structural in nature, relating the elements in the input model of the transformation to the produced output model.

The contract proving approach we follow determines whether the elements present in the contract are present in the path conditions produced. If so, then through the abstraction relation we know that these elements will also appear in the abstracted input-output models. If not, then the path condition represents a counter-example to the contract.

The counter-examples allows the user to reason about the interactions between rules and determine if the transformation behaviour is as expected. This contract verification approach allows us to reason about the infinite set of transformation input-output models by examining a finite set of path conditions.

9.1.3 Design of SyVOLT

Our fourth research question *What is the design and work-flow of a contract verification tool?* is addressed in Chapter 5.

In that chapter, we focus on providing relevant details for our implementation of the path condition building and contract proving processes. In particular, we highlight how our SyVOLT tool fits into the model-driven and multi-formalism paradigm with a presentation in Section 5.1 of a Formalism Transformation Graph and Process Model. This allows for the elegant expression of the work-flow of the tool.

Chapter 5 also presents the mechanisms for creating transformations and contracts in two modelling tools, Eclipse (Section 5.2.2) and MPS (Section 5.2.3). The creation of these plug-ins allows for our contract proving approach to be used by these two large research and development communities.

As well, we discuss a few low-level details of the SyVOLT tool in Section 5.3. These details include the representation of typed graphs within our tool which provides context for our scalability techniques, as well as the model manipulations we perform in the PyRamify component which provides motivation for reasoning about model transformations at a higher level.

9.1.4 Algorithmic Scalability Techniques

Chapter 6.3 discusses our solution to our last research question: *What are techniques for improving the scalability of the verification tool to allow for larger transformations to be verified?*

As discussed in that chapter, the production of a state-space for a transformation suffers from exponential resource requirements as the transformation size increases. Chapter 6.3 therefore details three algorithmic techniques for reducing these requirements. The first technique is to make the path condition generation and contract proving algorithm parallel such that multiple operations can take place at one time. Slicing is the second technique, wherein only those rules relevant to the contract being proved are retained in the transformation. Finally, we examined the pruning technique, where study of the output meta-model of the transformation reveals conditions where path conditions can be declared invalid and removed.

Results were provided for each of the techniques for our case studies (Chapter 7), displaying how they are beneficial to reducing the time taken by the contract prover. In particular, our numeric results indicate that our technique is scalable and widely applicable, as we can prove multiple contracts on relatively large transformations.

9.2 Future Work

This thesis has presented our contribution to contract verification for DSLTrans model transformations. This was achieved by providing an end-to-end approach, such that a) the precise semantics of DSLTrans was presented, b) the contract proving approach was discussed with examples, c) the SyVOLT tool was introduced, and d) the tool was shown to be scalable to our case studies.

This section will examine the future work possible in this research.

9.2.1 Transfer Contract Proving Approach

A large part of the current thesis has focused on the formalization of the DSLTrans language and the contract proving approach within the double-pushout approach. This approach was selected as it is common within the model transformation community, and multiple verification techniques and tools exist.

Therefore, a future direction of research is to transfer our contract proving approach to be applicable to other double-pushout/ graph transformation 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. An initial list would include restrictions like *termination*, *confluence*, *non-deletion*, and *layer-based*.

Extending our approach to a new language may be as simple as defining matchers and rewriters for the rules in the language, similar to that performed in Section 3.1.4. As well, any constructs in those languages not handled in the current thesis would have to be added to the path condition generation and contract proving techniques.

Further generalization to languages that allow deletion, non-termination, or other relaxations of expressiveness may be possible. However, as these restrictions are so core to our approach, an entirely new verification approach may need to be taken.

9.2.2 Creation Process for Contracts

As with other verification approaches, it is not trivial to determine which properties are of interest for a transformation. For our case studies, the contracts were non-systematically created to verify certain properties, which may miss interesting or erroneous behaviours in the transformation.

Future work will focus on determining a systematic way of creating contracts to fully verify a transformation. In particular, we will work with transformation developers (such as the creators of the *mbeddr* transformation in Section 7.5 on page 270) to understand what properties are of interest, and how these properties should be translated into contracts. One possible approach to contract generation is to have the transformation designer write sentences about the properties they wish to prove, and generate contracts directly from these sentences.

9.2.3 Integration with Modelling Tools

Another direction of our research is to integrate our contract proving approach with academic and industrial tools. In particular, we are interested in integrating our language into other model transformation engines such as VIATRA [25] and the ModelVerse [158, 160]. This integration would be made possible through the extension of our contract proving approach to the transformation languages defined in those tools, or by providing plug-ins such that DSLTrans transformations and contracts can be created. We are also interested in examining how our algorithmic scalability techniques and modular tool components can be transferred to other domains and modelling tools.

A critical aspect of model-driven engineering is to model everything explicitly and at the correct abstraction level. We have attempted to follow this principle in our approach to developing our SyVOLT tool. However, the PyRamify component breaks this principle, as it is encoded as Python scripts which use explicit manipulation of Python classes and variables to create the *Matchers* and *Rewriters* needed. This is due to the requirement to perform RAMification on the rule language, which is composed of the input language, the output language, and the DSLTrans language. To our knowledge, no other modelling tools are available to automatically perform this operation. Therefore it is our intention to work with modelling tool developers to natively enable this type of reasoning.

The eventual goal of the SyVOLT development process is to integrate the contract prover directly into model transformation building processes, such that contracts are continually checked on the transformation during development. This 'contract-driven' development could reduce the time to develop transformations by quickly detecting counter-examples. This would be similar to the continuous verification approach performed by Ratiu *et al.* in MPS [124].

Bibliography

[1] Apache Ant. `http://ant.apache.org`.

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

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

[4] Autosar. `https://www.autosar.org`.

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

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

[7] Graphical Modelling Framework. `https://eclipse.org/gmf-tooling/`.

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

[9] mbeddr. `http://mbeddr.com/`.

[10] RSS 2.0 Specification. `http://www.rssboard.org/rss-specification`.

[11] The igraph Network Analysis Package. `http://igraph.org/`.

[12] 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.

[13] M. Amrani, J. Dingel, L. Lambers, L. Lúcio, R. Salay, G. Selim, E. Syriani, and M. Wimmer. Towards a model transformation intent catalog. In *Proceedings of the First Workshop on Analysis of Model Transformations*, pages 3–8. ACM, Oct. 2012.

[14] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *International*

Conference on Model Driven Engineering Languages and Systems, pages 121–135. Springer, 2010.

[15] M. Asztalos, L. Lengyel, and T. Levendovszky. Towards automated, formal verification of model transformations. In *International Conference on Software Testing, Verification and Validation*, pages 15–24. IEEE, 2010.

[16] B. Azizi, B. Zamani, and S. Kolahdouz-Rahimi. Contract verification of ETL transformations. In *International Conference on Computer and Knowledge Engineering*, pages 154–160. IEEE, Oct. 2017.

[17] A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, et al. Workflow-driven tool integration using model transformations. In *Graph Transformations and Model-Driven Engineering*, pages 224–248. Springer Berlin Heidelberg, 2010.

[18] A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Proceedings of the Symposium on Applied Computing*, pages 1280–1287. ACM, 2006.

[19] B. Barroca. *Analysable software language translations*. PhD thesis, Faculdade de Ciências e Tecnologia, 2013.

[20] B. Barroca, V. Amaral, and D. Buchs. Semantic languages for developing correct language translations. *Software Quality Journal*, pages 1–37, Jan. 2017.

[21] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing incomplete transformation language. In *International Conference on Software Language Engineering*, pages 296–305. Springer Berlin Heidelberg, 2011.

[22] J. Bauer, I. Boneva, M. E. Kurbán, and A. Rensink. A modal-logic based graph abstraction. In *International Conference on Graph Transformation*, volume 5214 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2008.

[23] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *International Conference on Software Engineering*, pages 72–81. ACM, 2006.

[24] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - A tool suite for automatic verification of real-time systems. In *International Hybrid Systems Workshop*, pages 232–243. Springer, Springer, 1996.

[25] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. VIATRA 3: A reactive model transformation platform. In *International Conference on Theory and Practice of Model Transformations*, pages 101–110. Springer.

[26] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract-aware. *Computer*, 32(7):38–45, 1999.

[27] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[28] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? Transformation models! In *International Conference on Model Driven Engineering Languages and Systems*, volume 4199, pages 440–453. Springer, 2006.

[29] E. Biermann. Local confluence analysis of consistent EMF transformations. *Electronic Communications of the European Association for the Study of Science and Technology*, 38:68–84, 2011.

[30] E. Biermann, C. Ermel, and G. Taentzer. Precise semantics of EMF model transformations by graph transformation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 53–67. Springer, 2008.

[31] D. Bisztray and R. Heckel. Combining termination proofs in model transformation systems. *Mathematical Structures in Computer Science*, 24(4), 2014.

[32] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language reference manual*. Addison-Wesley, 1999.

[33] K. Born, T. Arendt, F. Hess, and G. Taentzer. Analyzing conflicts and dependencies of rule-based transformations in Henshin. In *International Conference on Fundamental Approaches to Software Engineering*, pages 165–168. Springer, Berlin, Heidelberg, 2015.

[34] M. Brambilla, J. Cabot, and M. Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[35] H. S. Bruggink. Towards a systematic method for proving termination of graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 213(1):23–38, 2008.

[36] L. Burgueno, J. Troya, M. Wimmer, and A. Vallecillo. Static fault localization in model transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, 2015.

[37] F. Büttner, M. Egea, and J. Cabot. On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems*, pages 432–448. Springer, 2012.

[38] F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL transformations using transformation models and model finders. In *International Conference on Formal Engineering Methods*, pages 198–213. Springer, 2012.

[39] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL contracts for the verification of model transformations. *Electronic Communications of the Electronic Communications of the European Association for the Study of Science and Technology*, 24, 2010.

[40] V. Carletti, P. Foggia, A. Saggese, and M. Vento. Introducing VF3: A new algorithm for subgraph isomorphism. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 128–139. Springer, 2017.

[41] R. Clarisó, J. Cabot, E. Guerra, and J. de Lara. Backwards reasoning for model transformations: Method and applications. *Journal of Systems and Software*, 116:113–132, 2016.

[42] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All about Maude - A high-performance logical framework: How to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[43] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *International Association for Pattern Recognition workshop on Graph-based Representations in Pattern Recognition*, pages 149–159, 2001.

[44] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)-graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[45] K. Czarnecki and S. Helsen. Classification of model transformation approaches. *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications Workshop*, 45(3):1–17, 2003.

[46] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139–163, 2007.

[47] J. de Lara and H. Vangheluwe. Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, 22(3-4):297–326, May 2010.

[48] M. De Santo, P. Foggia, C. Sansone, and M. Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079, May 2003.

[49] I. Dragomir, I. Ober, and C. Percebois. Contract-based modeling and verification of timed safety requirements within SysML. *Software and Systems Modeling*, 16(2):587–624, 2017.

[50] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer-Verlag New York, Inc., 2006.

[51] H. Ehrig, G. Taentzer, J. de Lara, D. Varró, and S. Varró-Gyapai. Termination criteria for model transformation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 49–63. Springer, 2005.

[52] R. Floyd. Algorithm 97: Shortest path. *Communications of the Association for Computing Machinery*, 5(6):345, 1962.

[53] M. Fowler. Language workbenches: The killer-app for domain specific languages? `https://martinfowler.com/articles/languageWorkbench.html`, June 2005.

[54] M. Fowler. Projectional editing. `https://martinfowler.com/bliki/ProjectionalEditing.html`, Jan. 2008.

[55] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[56] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo. EUnit: a unit testing framework for model management tasks. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems*, pages 395–409. Springer, 2011.

[57] L. Geiger and A. Zündorf. Tool modeling with Fujaba. *Electronic Notes in Theoretical Computer Science*, 148(1):173–186, 2006.

[58] C. Gerking, W. Schäfer, S. Dziwok, and C. Heinzemann. Domain-specific model checking for cyber-physical systems. In *Model Development, Validation and Verification workshop at International Conference on Model Driven Engineering Languages and Systems*, pages 18–27, 2015.

[59] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012.

[60] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards verified model transformations. In *Proceedings of the 3rd International Workshop on Model Development, Validation and Verification*, pages 78–93. Citeseer, 2006.

[61] H. Giese, S. Hildebrandt, and L. Lambers. Bridging the gap between formal semantics and implementation of triple graph grammars. *Software and Systems Modeling*, 13(1):273–299, 2014.

[62] P. Giner and V. Pelechano. Test-driven development of model transformations. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems*, pages 748–752. Springer, Berlin, Heidelberg, 2009.

[63] M. Gogolla and A. Vallecillo. Tractable model transformation testing. In *Proceedings of European Conference on Modelling Foundations and Applications*, pages 221–235. Springer, Berlin, Heidelberg, 2011.

[64] C. Gomes, B. Barroca, and V. Amaral. Classification of model transformation tools: Pattern matching techniques. In *Model-Driven Engineering Languages and Systems SE - 38*, volume 8767 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014.

[65] C. Gomes, B. Barroca, and V. Amaral. *DSLTrans User Manual*, Sept. 2016. `http://msdl.cs.mcgill.ca/people/levi/files/DSLTransManual.pdf`.

[66] C. González and J. Cabot. ATLTest: a white-box test generation approach for ATL transformations. In *Model Driven Engineering Languages and Systems*, pages 449–464. Springer, 2012.

[67] E. Guerra, J. de Lara, D. Kolovos, and R. Paige. A visual specification language for model-to-model transformations. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 119–126. IEEE, 2010.

[68] E. Guerra, J. de Lara, and F. Orejas. Pattern-based model-to-model transformation: Handling attribute conditions. In *International Conference on Theory and Practice of Model Transformations*, pages 83–99. Springer, 2009.

[69] 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. *Automated Software Engineering*, 20(1):5–46, 2013.

[70] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3, 4):287–313, 1996.

[71] A. Habel and K.-h. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[72] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *International Conference for Graph Transformations*, pages 161–176. Springer, 2002.

[73] International Organization for Standardization. ISO 26262: Road vehicles-functional safety. `https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en`, 2011.

[74] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[75] M. Jukšs, C. Verbrugge, and H. Vangheluwe. Transformations debugging transformations. In *MDEbug: Debugging in Model-Driven Engineering at International Conference on Model Driven Engineering Languages and Systems*, 2017.

[76] J. Kärnä, J.-P. Tolvanen, and S. Kelly. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications workshop on Domain-Specific Modeling*, 2009.

[77] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003.

[78] S. Kelly. Domain-specific modeling 76 cases of MDD that works. *MetaCase*, 2009.

[79] S. Kelly and J. Tolvanen. *Domain-specific modeling: Enabling full code generation.* John Wiley & Sons, 2008.

[80] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report TR-RI-07-284, University of Paderborn, 2007.

[81] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[82] D. Kolovos, R. Paige, L. Rose, and F. Polack. Unit testing model management operations. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 97–104. IEEE, 2008.

[83] B. König and V. Kozioura. Augur 2 - a new version of a tool for the analysis of graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 211:201–210, 2008.

[84] T. Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[85] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. Explicit transformation modeling. In *International Conference on Model Driven Engineering Languages and Systems*, pages 240–255. Springer.

[86] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: Design and analysis of algorithms*, volume 400. Benjamin/Cummings, 1994.

[87] J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006.

[88] L. Lambers, H. Ehrig, and F. Orejas. Efficient detection of conflicts in graph-based model transformation. *Electronic Notes in Computer Science*, 152:97–109, 2006.

[89] L. Lambers, H. Ehrig, U. Prange, and F. Orejas. Embedding and confluence of graph transformations with negative application conditions. In *International Conference on Graph Transformation*, pages 162–177. Springer, 2008.

[90] L. Lambers, S. Hildebrandt, H. Giese, and F. Orejas. Attribute handling for bidirectional model transformations: The triple graph grammar case. *Electronic Communications of the European Association of Software Science and Technology*, 49, 2012.

[91] B. LaShomb and E. Syriani. Re-engineering DSLTrans with T-Core. Technical Report SERG-2012-04, Department of Computer Science, University of Alabama, 2012.

[92] T. Levendovszky, L. Lengyel, G. Mezei, and H. Charaf. A systematic approach to metamodeling environments and model transformation systems in VMTS. *Electronic Notes in Theoretical Computer Science*, 127(1):65–75, 2005.

[93] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. Selim, E. Syriani, and M. Wimmer. Model transformation intents and their properties. *Software and Systems Modeling*, 15(3):1–38, 2015.

[94] L. Lúcio, B. Barroca, and V. Amaral. A technique for automatic validation of model transformations. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems*, pages 136–150. Springer, 2010.

[95] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. FTG+PM: An integrated framework for investigating model transformation chains. In *International System Design Languages Forum*, pages 182–202. Springer, 2013.

[96] L. Lúcio and B. Oakes. Language verification repository. `https://github.com/mbeddr/language_verification`.

[97] L. Lúcio, B. Oakes, C. Gomes, G. Selim, J. Dingel, J. Cordy, and H. Vangheluwe. SyVOLT: Full model transformation verification using contracts. In *International Conference on Model Driven Engineering Languages and Systems*, pages 24–27, 2015.

[98] L. Lúcio, B. Oakes, and H. Vangheluwe. A technique for symbolically verifying properties of graph-based model transformations. Technical Report SOCS-TR-2014.1, McGill University, 2014.

[99] L. Lúcio and H. Vangheluwe. Model transformations to verify model transformations. In *Proceedings of the Workshop on Verification of Model Transformations*, 2013.

[100] R. Mannadiar. *A multi-paradigm modelling approach to the foundations of domain-specific modelling.* PhD thesis, McGill University, 2012.

[101] R. Mannadiar and H. Vangheluwe. Debugging in domain-specific modelling. In *Proceedings of Software Language Engineering*, pages 276–285. Springer, 2010.

[102] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[103] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer. Promobox: A framework for generating domain-specific property languages. In *International Conference on Software Language Engineering*, pages 1–20. Springer, 2014.

[104] P. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004.

[105] J. Mottu, B. Baudry, and Y. Le Traon. Model transformation testing: Oracle issue. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 105–112, 2008.

[106] S. Mustafiz, J. Denil, L. Lúcio, and H. Vangheluwe. The FTG+PM framework for multi-paradigm modelling: An automotive case study. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pages 13–18. ACM, 2012.

[107] A. Narayanan and G. Karsai. Verifying model transformations by structural correspondence. *Electronic Communications of the European Association of Software Science and Technology*, 10, 2008.

[108] M. Nottingham and R. Sayre. The ATOM syndication format. RFC 4287, Network Working Group, Dec. 2005.

[109] B. Oakes. EcoreImport project. `https://github.com/BentleyJOakes/EcoreImport`, 2017.

[110] B. Oakes, L. Lúcio, C. Gomes, and H. Vangheluwe. Expressive symbolic-execution contract proving for the DSLTrans transformation language. Technical Report 2017-01, McGill University, 2017.

[111] B. Oakes, L. Lúcio, C. Gomes, and H. Vangheluwe. Semantics of the DSLTrans transformation language via the double-pushout approach. In preparation, 2018.

[112] B. Oakes, J. Troya, L. Lúcio, and M. Wimmer. Fully verifying transformation contracts for declarative ATL. In *International Conference on Model Driven Engineering Languages and Systems*, pages 256–265, 2015.

[113] B. Oakes, J. Troya, L. Lúcio, and M. Wimmer. Full contract verification for ATL using symbolic execution. *Software and Systems Modeling*, pages 1–35, 2016.

[114] B. Oakes, C. Verbrugge, L. Lúcio, and H. Vangheluwe. Debugging of model transformations and contracts in SyVOLT. In *MDEbug: Debugging in Model-Driven Engineering at International Conference on Model Driven Engineering Languages and Systems*, 2018.

[115] Object Management Group. *OMG Unified Modeling Language - Version 2.5*, Mar. 2015.

[116] F. Orejas and L. Lambers. Symbolic attributed graphs for attributed graph transformation. *Electronic Communications of the European Association of Software Science and Technology*, 30, 2010.

[117] E. Paen. Measuring incrementally developed model transformations using change metrics. Master's thesis, Queen's University, 2012.

[118] D. Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

[119] D. Plump. Confluence of graph transformation revisited. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, pages 280–308. Springer, 2005.

[120] E. Posse and J. Dingel. Kiltera: A language for timed, event-driven, mobile and distributed simulation. In *International Symposium on Distributed Simulation and Real Time Applications*, pages 87–96. IEEE, 2010.

[121] E. Posse and J. Dingel. An executable formal semantics for UML-RT. *Software and Systems Modeling*, pages 1–39, 2014.

[122] M. Provost. Himesis: A hierarchical subgraph matching kernel for model driven development. Master's thesis, McGill University, 2005.

[123] L. A. Rahim and J. Whittle. A survey of approaches for verifying model transformations. *Software and Systems Modeling*, 14(2):1003–1028, 2015.

[124] D. Ratiu, B. Schaetz, M. Voelter, and B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, pages 9–15. IEEE Press, 2012.

[125] A. Rensink and D. Distefano. Abstract graph transformation. *Electronic Notes in Theoretical Computer Science*, 157(1):39–59.

[126] A. Rensink, A. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *International Conference on Graph Transformations*, pages 226–241. Springer, 2004.

[127] A. Rensink and E. Zambon. Pattern-based graph abstraction. In *International Conference on Graph Transformation*, pages 66–80. Springer, 2012.

[128] J. Rivera, F. Durán, and A. Vallecillo. Formal specification and analysis of domain specific models using Maude. *Simulation*, 85(11-12):778–792, 2009.

[129] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with Maude. *Software Language Engineering*, 5452:54–73, 2008.

[130] L. Rose, R. Paige, D. Kolovos, and F. Polack. The Epsilon Generation Language. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture - Foundations and Applications SE - 1*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.

[131] O. Runge, C. Ermel, and G. Taentzer. AGG 2.0 - new features for specifying and analyzing algebraic graph transformations. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, pages 81–88. Springer, 2011.

[132] B. Schätz. Verification of model transformations. *Electronic Communications of the European Association of Software Science and Technology*, 29, 2010.

[133] D. Schmidt. Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[134] A. Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.

[135] G. Selim. *Formal Verification of Graph-Based Model Transformations*. PhD thesis, Queen's University, 2015.

[136] G. Selim, J. Cordy, J. Dingel, L. Lúcio, and B. Oakes. Finding and fixing bugs in model transformations with formal verification: An experience report. In *Proceedings of Analysis of Model Transformations workshop at Model Driven Engineering Languages and Systems*, pages 26–35, 2015.

[137] G. Selim, L. Lúcio, J. Cordy, J. Dingel, and B. Oakes. Specification and verification of graph-based model transformation properties. In *Proceedings of International Conference on Graph Transformation*, pages 113–129. Springer, 2014.

[138] G. Selim, S. Wang, J. Cordy, and J. Dingel. Model transformations for migrating legacy models: An industrial case study. In *Proceedings of European Conference on Modelling Foundations and Applications*, pages 90–101. Springer, 2012.

[139] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model pruning. In *International Conference on Model Driven Engineering Languages and Systems*, pages 32–46. Springer, 2009.

[140] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[141] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[142] E. Syriani. *A multi-paradigm foundation for model transformation language engineering*. PhD thesis, McGill University, 2011.

[143] E. Syriani and H. Vangheluwe. Matters of model transformation. Technical Report SOCS-TR-2009.2, School of Computer Science, McGill University, 2009.

[144] E. Syriani and H. Vangheluwe. Performance analysis of Himesis. Technical Report SOCS-TR-2010.8, School of Computer Science, McGill University, 2010.

[145] E. Syriani, H. Vangheluwe, and B. LaShomb. T-Core: A framework for custom-built model transformation engines. *Software & Systems Modeling*, 14(3):1215–1243, 2015.

[146] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. AToMPM: A web-based modeling environment. In *International Conference on Model Driven Engineering Languages and Systems*, pages 21–25, 2013.

[147] G. Taentzer. AGG: A tool environment for algebraic graph transformation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, volume 1779, pages 333–341. Springer, 2000.

[148] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. *Proceedings of the Workshop for Model Transformation in Practice at the International Conference on Model Driven Engineering Languages and Systems*, 2005.

[149] F. Tip. *A survey of program slicing techniques.* Centrum voor Wiskunde en Informatica, 1994.

[150] M. Tisi, S. Martínez, F. Jouault, and J. Cabot. Refining models with rule-based model transformations. Research Report RR-7582, INRIA, 2011.

[151] J. Troya and A. Vallecillo. A rewriting logic semantics for atl. *Journal of Object Technology*, 10(5):1–29, 2011.

[152] Z. Ujhelyi. *Program Analysis Techniques for Model Queries and Transformations.* PhD thesis, Budapest University of Technology and Economics, 2016.

[153] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015.

[154] A. Vallecillo, M. Gogolla, L. Burgueno, M. Wimmer, and L. Hamann. Formal specification and testing of model transformations. In *Formal Methods for Model-Driven Engineering*, pages 399–437, 2012.

[155] M. F. van Amstel. *Assessing and improving the quality of model transformations.* PhD thesis, Technische Universiteit Eindhoven, 2012.

[156] H. Van der Auweraer, J. Anthonis, S. De Bruyne, and J. Leuridan. Virtual engineering at work: The challenges for designing mechatronic products. *Engineering with Computers*, 29(3):389–408, 2013.

[157] S. Van Mierlo. Explicitly modelling model debugging environments. In *International Conference on Model Driven Engineering Languages and Systems*, pages 24–29, 2015.

[158] S. Van Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, and T. Kühne. Multi-level modelling in the modelverse. In *International Conference on Model Driven Engineering Languages and Systems*, pages 83–92, 2014.

[159] S. Van Mierlo, Y. Van Tendeloo, B. Meyers, and H. Vangheluwe. Domain-specific modelling for human–computer interaction. In *The Handbook of Formal Methods in Human-Computer Interaction*, pages 435–463. Springer International Publishing, 2017.

[160] Y. Van Tendeloo. Foundations of a multi-paradigm modelling tool. In *International Conference on Model Driven Engineering Languages and Systems*, pages 52–57, 2015.

[161] H. Vangheluwe. Foundations of modelling and simulation of complex systems. *Electronic Communications of the EASST*, 10, 2008.

[162] H. Vangheluwe, J. de Lara, and P. J. Mosterman. An introduction to multi-paradigm modelling and simulation. In *Proceedings of AI, Simulation and Planning in High Autonomy Systems*, pages 9–20. SCS, 2002.

[163] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and Systems Modeling*, 3(2):85–113, 2004.

[164] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *International Conference on the Unified Modeling Language*, pages 290–304. Springer, 2004.

[165] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, 2002.

[166] D. Varró, S. Varró-Gyapai, H. Ehrig, U. Prange, and G. Taentzer. Termination analysis of model transformations by Petri nets. In *International Conference on Graph Transformation*, volume 4178, pages 260–274. Springer, 2006.

[167] G. Varró, K. Friedl, and D. Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electronic Notes in Theoretical Computer Science*, 152:191–205, 2006.

[168] M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. *Using C language extensions for developing embedded software: A case study*, volume 50. ACM, 2015.

[169] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: An extensible C-based programming language and IDE for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[170] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Professional, 1998.

[171] S. Warshall. A theorem on boolean matrices. *Journal of the Association for Computing Machinery*, 9(1):11–12, 1962.

[172] M. Wieber, A. Anjorin, and A. Schürr. On the usage of TGGs for automated model transformation testing. In *International Conference on Theory and Practice of Model Transformations*, pages 1–16. Springer, 2014.