FORMAL VERIFICATION OF GRAPH-BASED MODEL TRANSFORMATIONS

by

GEHAN MUSTAFA KAMEL SELIM

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
June 2015

Copyright © Gehan Mustafa Kamel Selim, 2015
Abstract

Model Driven Development (MDD) is a relatively new software development methodology that has been increasingly used in the last decade for software development and, in many cases, has replaced traditional, code-centric approaches. In MDD, *models* or software abstractions are the basic building blocks in the software development life cycle and *model transformations* are the technology used to map between models conforming to different metamodels. Model transformations are used for different purposes in MDD, e.g., refactoring, migration, and code generation. Since model transformations are essential in MDD, transformation testing and verification is essential to the success of MDD and has been of increasing interest to researchers and practitioners.

In this research, we investigate the verification of model transformations with respect to a wide range of properties in an automatic and scalable fashion using symbolic execution techniques. First, we survey the state-of-the-art in testing and verification of model transformations. Second, we present a model transformation that we have previously developed in an industrial context and used later on as a case study for experimentation. Third, we experiment with a black-box testing tool and an automated formal verification tool on the aforementioned industrial case study. This step was intended to give us a better understanding of the limitations of current tools.
that yet need to be addressed by researchers. Fourth, we attempt to address the limitations encountered in the state-of-the-art tools by extending and enhancing a symbolic model transformation property prover for a graph-based transformation language called DSLTrans. Finally, we use our symbolic model transformation property prover to verify properties for our industrial transformation and for another large transformation, both of which we reimplemented in DSLTrans. We report on the results, strengths and limitations of our property prover in comparison with other verification tools, lessons learnt, and possible future work.
Co-Authorship

All papers resulting from this thesis were co-authored with my supervisors Dr. James R. Cordy and Dr. Juergen Dingel. Some of these papers were co-authored with other collaborators. In all such papers, I am the primary author.

Part of Chapter 2 was a technical report [137] and another part of Chapter 2 was published in the proceedings of the Analysis of Model Transformations (AMT’12) workshop [138], where both papers were co-authored with James R. Cordy and Juergen Dingel. Part of Chapter 3 was published in the proceedings of the European Conference on Modelling Foundations and Applications (ECMFA’12) co-authored with Shige Wang, James R. Cordy, and Juergen Dingel [141], and received the Best Paper Award. Part of Chapter 4 in Section 4.1 was published in the Software and Systems Modeling Journal (SoSym’15) coauthored with Shige Wang, James R. Cordy, and Juergen Dingel [142]. Part of Chapter 4 in Section 4.2 was published in the proceedings of the Conference on Model Driven Engineering Languages and Systems (MODELS’13) coauthored with Fabian Büttner, James R. Cordy, Juergen Dingel, and Shige Wang [136], and was nominated for the Best Paper Award. Part of Chapter 5 was published in the proceedings of the International Conference on Graph Transformation (ICGT’14) co-authored with Levi Lúcio, James R. Cordy, Juergen Dingel, and Bentley J. Oakes [140]. Chapter 5 was also partially based on a technical
report co-authored with Levi Lúcio, James R. Cordy, and Juergen Dingel [139]. Part of Chapter 6 in Section 6.1 was published in the proceedings of the International Conference on Graph Transformation (ICGT’14) co-authored with Levi Lúcio, James R. Cordy, Juergen Dingel, and Bentley J. Oakes [140].
Acknowledgments

In the name of God, the Most Gracious the Most Merciful. Endless praises and gratitude are due to He who guided me through my life and career. It is only with God's blessings, sustenance, and bestowal of constant support that I was able to finalize my PhD degree.

My sincere gratitude and appreciation are due to my mentors and supervisors Prof. James R. Cordy and Prof. Juergen Dingel. Words cannot express how much I have learnt from you on the personal and professional levels. Providing me with the perfect work environment, the brainstorming sessions, and your advices on all matters are just a few of the things you have given me and which helped develop my personal and professional skills. You have been very patient and supportive throughout my PhD journey and you have made it a very fruitful experience. It has been a true honor and pleasure to work with you.

Mum and Reham, this is to you. To a life-time of support and love, to the encouragement you have given me since the day I received the PhD offer, and to the times when I felt home sick and you talked me through it. Nothing of this would have been possible without you. I owe you my happiness and success. My dear nephew Yassin, this degree is one of many milestones I plan to achieve in hope of being a good example for you. Your arrival to this world two years ago made life a pure bliss.
Dad, while you have left us a while ago and while you were not physically with me throughout this journey, you have always been in my heart and mind. Everything you have taught me and all the love you have surrounded me with will stay with me for life. I hope that I made you proud.

My fiance and soul mate, Mohamed, I will not even attempt to express all what I would like to say since that will require a separate thesis. I hope you understand how much you changed my perspective of life, love, trust, and loyalty. While being a PhD student is difficult, being engaged to one and understanding their unstable work schedules is not at all easy. You have walked me through this bumpy journey with so much patience and support. I will always do my best to be the partner and life companion you deserve to have. To my new family in Egypt especially Nagat, your prayers and encouragement are invaluable and mean the world to me. Thank you for the upbringing of the man of my dreams.

I would like to thank the STL Lab members (Karolina, Charu, Scott, Manar, Douglas, Eric, Andrew, Matthew, Eyrak, Aseel, Mark, Amal, Boris, Suchita, Nondini, Leo, David, and Tuhin) for being amazing colleagues and friends. All the prank wars, the jokes, and any work we conducted together define for me how lab members should be like with each other.

I am indebted to my best friends Heidi, Nahla, Yara, and Shereen from Egypt for their unconditional support even when we were continents apart. You have tolerated my tantrums, mood swings, and you have been there for me in good and bad times. I am lucky to have crossed your path in life and I am even more lucky to be such a close friend to you. I would also like to thank my friends from Informateks, Egypt (Heidi, Nahla, Yara, Marwa, Tawfik, Ramez, Garhy, Engy, Suwaify, Bayoumi, Diab,
Shebl, Ibrahim, Salah); we had so many inside jokes and laughs that are enough to keep me smiling forever.

Since I arrived in Canada, I have had the best company any one can ask for. Lili Barbour, you were the first friend I made and you have walked me through many situations knowing that I am a foreigner and that I can use the extra help. Thank you for being there for me. Rana, Mariam, Hisham, and Ramy you all have enlightened the last years of my PhD. While I did not know you for long, I truly consider you my family in Kingston. To my dear friends Shereen, Layan, Razan, Khadijah, Dina, Alsaadis, Mervat, Wissam, Ghadeer, Inas, Khaled, Mahmoud, Amr, Sherief, Ashraf, Anas and Taher thank you for the trips, celebrations, and the good times we had together.

Finally, I would like to acknowledge everyone who stepped in my life at some point and gave me an advice, a smile, a kind word, or even a rough time. I have learnt a lot over the years, and I am looking forward to learn even more.
Statement of Originality

I, Gehan Mustafa Kamel Selim, hereby certify that all of the work described within this thesis is the original work of the author. The research was conducted under the supervision of Dr. James R. Cordy and Dr. Juergen Dingel. Any published (or unpublished) ideas and/or techniques of others are fully acknowledged in accordance with the standard referencing practices.

Gehan Mustafa Kamel Selim, June 2015
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEM Criterion</td>
<td>Association-End Multiplicity Criterion</td>
</tr>
<tr>
<td>AMP Criterion</td>
<td>All Message Paths Criterion</td>
</tr>
<tr>
<td>ASP</td>
<td>Answer Set Programming</td>
</tr>
<tr>
<td>AToM3</td>
<td>A Tool for Multi-formalism and Meta-Modelling</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>AUTomotive Open System ARchitecture</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
</tr>
<tr>
<td>BON</td>
<td>Business Object Notation</td>
</tr>
<tr>
<td>CA Criterion</td>
<td>Class Attribute Criterion</td>
</tr>
<tr>
<td>CNF</td>
<td>Conjunctive Normal Form</td>
</tr>
<tr>
<td>Coll Criterion</td>
<td>Collection Coverage Criterion</td>
</tr>
<tr>
<td>Cond Criterion</td>
<td>Condition Coverage Criterion</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>DPO</td>
<td>Double Pushout</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>EHA</td>
<td>Extended Hybrid Automata</td>
</tr>
<tr>
<td>EML Criterion</td>
<td>Each Message on Link Criterion</td>
</tr>
<tr>
<td>EVL</td>
<td>Epsilon Validation Language</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>FP Criterion</td>
<td>Full Predicate Coverage Criterion</td>
</tr>
<tr>
<td>GM</td>
<td>General Motors</td>
</tr>
<tr>
<td>GN Criterion</td>
<td>GeNeralization Criterion</td>
</tr>
<tr>
<td>GReAT</td>
<td>Graph Rewriting and Transformation</td>
</tr>
<tr>
<td>GROOVE</td>
<td>GRaphs for Object-Oriented VERification</td>
</tr>
<tr>
<td>GRS</td>
<td>Graph Rewriting System</td>
</tr>
<tr>
<td>HLRU</td>
<td>High Level Replacement Unit</td>
</tr>
<tr>
<td>HOT</td>
<td>Higher Order Transformation</td>
</tr>
<tr>
<td>JAST</td>
<td>Java Abstract Syntax Tree</td>
</tr>
<tr>
<td>JML</td>
<td>Java Modeling Language</td>
</tr>
<tr>
<td>JPF</td>
<td>Java Path Finder</td>
</tr>
<tr>
<td>KIV</td>
<td>Karlsruhe Interactive Verifier</td>
</tr>
<tr>
<td>LHS</td>
<td>Left Hand Side</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>MDD</td>
<td>Model Driven Development</td>
</tr>
<tr>
<td>MEL</td>
<td>Membership Equational Logic</td>
</tr>
<tr>
<td>MMCC</td>
<td>MetaModel Coverage Checker</td>
</tr>
<tr>
<td>MONA Solver</td>
<td>MONAdic second-order logic Solver</td>
</tr>
<tr>
<td>MSO</td>
<td>Monadic Second Order Logic</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-View Consistency Checking</td>
</tr>
<tr>
<td>NAC</td>
<td>Negative Application Condition</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PAC</td>
<td>Positive Application Condition</td>
</tr>
<tr>
<td><strong>PACO-Checker</strong></td>
<td>PaMoMo Contract-Checker</td>
</tr>
<tr>
<td><strong>PaMoMo</strong></td>
<td>Pattern-based Modeling Language for Model Transformations</td>
</tr>
<tr>
<td><strong>Promela</strong></td>
<td>Process Meta Language</td>
</tr>
<tr>
<td><strong>PVS</strong></td>
<td>Prototype Verification System</td>
</tr>
<tr>
<td><strong>QVT</strong></td>
<td>Query/View/Transformation</td>
</tr>
<tr>
<td><strong>RHS</strong></td>
<td>Right Hand Side</td>
</tr>
<tr>
<td><strong>SAT Solver</strong></td>
<td>SATisfiablity Solver</td>
</tr>
<tr>
<td><strong>SMT</strong></td>
<td>Satisfiability Modulo Theories</td>
</tr>
<tr>
<td><strong>SPIN</strong></td>
<td>Simple Promela INterpreter</td>
</tr>
<tr>
<td><strong>SPO</strong></td>
<td>Single Pushout</td>
</tr>
<tr>
<td><strong>TGG</strong></td>
<td>Triple Graph Grammar</td>
</tr>
<tr>
<td><strong>TTPN</strong></td>
<td>Timed Transition Petri Net</td>
</tr>
<tr>
<td><strong>UML</strong></td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td><strong>UML-RT</strong></td>
<td>Unified Modelling Language - Real Time</td>
</tr>
<tr>
<td><strong>USE</strong></td>
<td>UML-based Specification Environment</td>
</tr>
<tr>
<td><strong>VCS</strong></td>
<td>Vehicle Control Software</td>
</tr>
<tr>
<td><strong>VMTS</strong></td>
<td>Visual Modeling and Transformation System</td>
</tr>
</tbody>
</table>
# Contents

Abstract  
Co-Authorship  
Acknowledgments  
Statement of Originality  
List of Acronyms  
Contents  
List of Tables  
List of Figures

## Chapter 1: Introduction

1.1 Problem and Thesis Statement  
1.2 Original Contributions  
1.3 Organization of Thesis

## Chapter 2: Related Work

2.1 A Taxonomy of Model Transformation Verification Techniques  
2.2 Static Verification Techniques  
  2.2.1 Type I Formal Methods  
  2.2.2 Type II Formal Methods  
2.3 Dynamic Verification Techniques  
  2.3.1 Type III Formal Methods  
  2.3.2 Model Checking  
  2.3.3 Design Space Exploration (DSE)  
  2.3.4 Instrumentation  
  2.3.5 Testing
List of Tables

2.1 Classification of formal methods used to verify model transformations. 9

3.1 Matched rules, their corresponding rules from Section 3.3.1, and their functionality. 53

3.2 Functional helpers, their corresponding rules from Section 3.3.1, and their functionality. 54

4.1 The types of ATL constructs used to reimplement the transformation, their designated names, and their input and output element types. 69

4.3 The two rules that required updates to address the two violations of multiplicity invariants. 74

4.4 Translation/Constraint Solving times (seconds) for the 18 constraints on different scopes. For a scope of 12, the verification of S1 did not terminate in a week. 75

4.2 Formulated OCL Constraints 82

6.1 The rules in each layer of the GM-to-AUTOSAR model transformation after reimplementing the transformation in DSLTrans, and their input and output types. 107
6.2 Time taken (in seconds) to verify the properties in Table 4.2 using our property prover (first row) and using a tool based on model finders that we experimented with in Section 4.2 (second row).

6.3 The rules in each layer of the UML-RT-to-Kiltera transformation after reimplementing the transformation in DSLTrans, their input and output types, their corresponding rules from Section 6.2.2.

6.4 Properties of interest for the UML-RT-to-Kiltera transformation.

6.5 Time taken (in seconds) to verify the multiplicity invariants shown in Table 6.4 using our property prover.

6.6 Time taken (in seconds) to verify the syntactic invariants and the pattern contracts shown in Table 6.4 using our property prover.

6.7 Time taken (in seconds) to verify the reachability properties shown in Table 6.4 using our property prover.
List of Figures

2.1 Taxonomy overview of model transformation verification techniques . 8

3.1 V-Diagram for the VCS development process. . . . . . . . . . . . . 45
3.2 The subset of the GM metamodel used in our transformation. . . . 47
3.3 The AUTOSAR System Template containing relevant types used by
our transformation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
3.4 (a) Sample GM input model and (b) its corresponding AUTOSAR
output model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.1 Transformation model example . . . . . . . . . . . . . . . . . . . . . 66
4.2 The tool chain used to perform the transformation verification. . . 67
4.3 The counterexample generated for the CompositionType_component
multiplicity invariant. . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

5.1 Household Language . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.2 Community Language . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.3 The Persons Transformation expressed in DSLTrans. . . . . . . . 88
5.4 Contract1; should hold. . . . . . . . . . . . . . . . . . . . . . . . . 92
5.5 Contract2; should not hold. . . . . . . . . . . . . . . . . . . . . . . 92
5.6 The architecture of our symbolic model transformation property prover. 93
5.7 Generation of the set of path conditions in iterations. .......................... 95
5.8 A path condition containing four combined rules from the Persons transformation (i.e., ‘HouseholdsToCommunity’, ‘FatherToMan’, ‘MotherToWoman’, ‘BuildCommunity’). ............................................. 97
5.9 Three AtomicContracts (i.e., cont1, cont2, and cont3) that can be used with different propositional operators to convey different properties for the Persons model transformation. ................................. 102

6.1 AtomicContract AC1 that is used to express property P1. ...................... 108
6.2 AtomicContracts AC2, AC3, and AC4 that are used to express property M6 as AC2 \implies_t_c (AC3 \land_t_c \neg_t_c AC4). ................................. 109
6.3 AtomicContracts AC5 and AC6 that are used to express property S1 as AC5 \implies_t_c AC6. ................................................................. 110
6.4 An exemplar UML-RT capsule diagram [120]. ................................. 115
6.5 An exemplar UML-RT state machine diagram [120]. ......................... 117
6.6 The syntax of Kiltera, as demonstrated by Posse and Dingel [120]. .... 120
6.7 Layers 1-3 of the initial UML-RT-to-Kiltera transformation implementation (figure continued in the next page). ................................. 130
6.8 AtomicContracts AC1, AC2, and AC3 that are used to express MM1 (Table 6.4) as AC1 \implies_t_c (AC2 \land_t_c \neg_t_c AC3). ................................. 143
6.9 AtomicContracts AC4 and AC5 that are used to express MM5 (Table 6.4) as AC4 \implies_t_c AC5. ................................................................. 143
6.10 AtomicContracts AC6, AC7, and AC8 that are used to express MM8 (Table 6.4) as $AC6 \implies tc ((AC7 \land tc \neg tc AC8) \lor tc (\neg tc AC7))$.

6.11 AtomicContracts AC9 and AC10 that are used to express MM11 (Table 6.4) as $AC9 \implies tc AC10$.

6.12 AtomicContracts AC11 and AC12 that are used to express SS1 (Table 6.4) as $AC11 \implies tc AC12$.

6.13 AtomicContracts AC13, AC14, AC15, and AC16 that are used to express SS3 (Table 6.4) as $(AC13 \land tc \neg tc AC14) \implies tc (AC15 \land tc \neg tc AC16)$.

6.14 AtomicContract AC17 that is used to express PP1 (Table 6.4).

6.15 Layers 1-3 of the updated UML-RT-to-Kiltera transformation implementation (figure continued in the next page).

C.1 AtomicContracts AC1, AC2, and AC3 that are used to express MM1 (Table 6.4) as $AC1 \implies tc (AC2 \land tc \neg tc AC3)$.

C.2 AtomicContracts AC18, AC19, and AC20 that are used to express MM2 (Table 6.4) as $AC18 \implies tc (AC19 \land tc \neg tc AC20)$.

C.3 AtomicContracts AC21, AC22, and AC23 that are used to express MM3 (Table 6.4) as $AC21 \implies tc (AC22 \land tc \neg tc AC23)$.

C.4 AtomicContracts AC24, AC25, and AC26 that are used to express MM4 (Table 6.4) as $AC24 \implies tc (AC25 \land tc \neg tc AC26)$.

C.5 AtomicContracts AC4 and AC5 that are used to express MM5 (Table 6.4) as $AC4 \implies tc AC5$. 
C.6 AtomicContracts AC27 and AC28 that are used to express MM6 (Table 6.4) as $AC27 \Rightarrow_{tc} AC28$. ................. 203
C.7 AtomicContracts AC29 and AC30 that are used to express MM7 (Table 6.4) as $AC29 \Rightarrow_{tc} AC30$. ................. 203
C.8 AtomicContracts AC6, AC7, and AC8 that are used to express MM8 (Table 6.4) as $AC6 \Rightarrow_{tc} ((AC7 \land_{tc} \neg_{tc} AC8) \lor_{tc} (\neg_{tc} AC7))$. ................. 204
C.9 AtomicContracts AC33, AC34, and AC35 that are used to express MM9 (Table 6.4) as $AC33 \Rightarrow_{tc} ((AC34 \land_{tc} \neg_{tc} AC35) \lor_{tc} (\neg_{tc} AC34))$. 204
C.10 AtomicContracts AC36, AC37, and AC38 that are used to express MM10 (Table 6.4) as $AC36 \Rightarrow_{tc} ((AC37 \land_{tc} \neg_{tc} AC38) \lor_{tc} (\neg_{tc} AC37))$. 204
C.11 AtomicContracts AC9 and AC10 that are used to express MM11 (Table 6.4) as $AC9 \Rightarrow_{tc} AC10$. ......................... 205
C.12 AtomicContracts AC11 and AC12 that are used to express SS1 (Table 6.4) as $AC11 \Rightarrow_{tc} AC12$. ......................... 205
C.13 AtomicContracts AC39 and AC40 that are used to express SS2 (Table 6.4) as $AC39 \Rightarrow_{tc} AC40$. ......................... 205
C.14 AtomicContracts AC13, AC14, AC15, and AC16 that are used to express SS3 (Table 6.4) as $(AC13 \land_{tc} \neg_{tc} AC14) \Rightarrow_{tc} (AC15 \land_{tc} \neg_{tc} AC16)$.206
C.15 AtomicContract AC17 that is used to express PP1 (Table 6.4). . . . 206
C.16 AtomicContract AC41 that is used to express PP2 (Table 6.4). . . . 207
C.17 AtomicContract AC42 that is used to express PP3 (Table 6.4). . . . 207
C.18 AtomicContract AC43 that is used to express PP4 (Table 6.4). . . . 208
C.19 AtomicContract AC44 that is used to express PP5 (Table 6.4). . . . 208
D.1 The source UML-RT metamodel of the UML-RT-to-Kiltera model transformation. 210
D.2 The target Kiltera metamodel of the UML-RT-to-Kiltera model transformation. 211
Chapter 1

Introduction

Model Driven Development (MDD) [25] is a relatively new software development methodology that uses models (i.e., software abstractions) as the central means for software specification and communication. In MDD, the software development process can be conceptually treated as a sequence of model transformations. In code generation for example, abstract models are successively transformed into detailed models, and eventually into code. Thus, model transformations are a key technology in MDD that can automate the entire software development process.

A model transformation is a program that maps one or more input models (conforming to a source metamodel) to one or more output models (conforming to a target metamodel). Mens and Van Gorp [98] proposed a multi-dimensional taxonomy of model transformations where several factors were used to classify model transformations. For instance, the heterogeneity of the manipulated metamodels qualifies a transformation as exogenous or endogenous. A transformation is exogenous if it manipulates different source and target metamodels, and is endogenous otherwise. The abstraction levels of the manipulated models qualify a transformation as a horizontal or a vertical transformation. A transformation is horizontal if the input and
output models are at the same abstraction level, and is vertical otherwise. Conservation of the input model qualifies a transformation as an in-place or an out-place transformation. A transformation is in-place if it directly alters the input model, and is out-place otherwise. The transformation’s arity or the number of models manipulated by a transformation is another factor that can be used to classify model transformations.

Since model transformations are usually intended to be repeatedly used for a class of models, it is important to develop effective techniques for model transformation verification and analysis [6]. Several studies discuss different model transformation verification approaches and surveys of the proposed approaches have been conducted [137, 138, 5, 3].

This thesis investigates the verification of properties of graph-based model transformations. To do so, we first conduct a literature review of the available model transformation verification techniques and tools. Then, we develop an industrial transformation that is intended to be used as a case study to experiment with two state-of-the-art tools. Experimenting with existing tools is intended to give us hands-on experience and a better understanding of the limitations of these tools. Based on the conducted literature review and based on our experimentation with available tools, we redevelop a symbolic model transformation property prover for model transformations implemented in the DSLTrans model transformation language [18]. DSLTrans is a Turing-incomplete graphical model transformation language that guarantees termination and confluence by construction. Our symbolic model transformation property prover is based on building the set of path conditions (i.e., possible symbolic executions) of a DSLTrans transformation and checking each of these path conditions for
some property. The property prover returns \textit{true} if the property of interest holds for all path conditions and returns \textit{false} if the property of interest does not hold for at least one path condition. Thus, our property prover is \textit{input-independent} [137, 5] and its verification result holds for the transformation when run on any input model.

To evaluate our property prover, we use it to verify properties for two transformations (including the above mentioned industrial transformation), and we report on the verification and performance results.

1.1 Problem and Thesis Statement

In this thesis, we formulate and focus on the following research question: “How can we efficiently verify properties of transformations expressed as input-output model relations?”. We focus on properties expressed as input-output model relations since they have been highly investigated in the literature, using both textual (e.g., [35, 6]) and graphical (e.g., [13, 148, 129]) property languages.

In an attempt to answer the above research question, we investigate verifying properties of transformations implemented in the graph-based model transformation language DSLTrans [18]. We redevelop a symbolic model transformation property prover for DSLTrans [93, 91] that was previously limited to verifying atomic contracts (i.e., constraints on input-output model relations). The improved symbolic model transformation property prover we present in this thesis supports a more expressive property language that facilitates verifying atomic contracts and compositions of atomic contracts in the form of propositional logic formulae.

\textbf{Thesis Statement:} Graph-based model transformations can be verified with respect to a wide range of properties in an input-independent, automatic, and scalable
fashion using symbolic execution techniques.

1.2 Original Contributions

The proposed research aims to improve on the current state-of-the-art in model transformation verification by developing a verification tool that addresses the limitations of existing model transformation verification techniques and tools. More specifically, the two main contributions of this study are:

1. We develop an industrial model transformation to migrate legacy models for General Motors (GM) in to their corresponding AUTOSAR models.

2. We use this industrial transformation to experiment with existing tools and learn more about their limitations, and we build our own verification tool in an attempt to address limitations of existing tools. More concretely, our tool performs input-independent and unbounded verification of many property types on a sound and complete representation of a transformation’s execution. Verification is performed on transformations implemented in an intuitive graphical formalism, and the scalability of our tool is demonstrated on two case studies.

We conduct the following steps to achieve the two above mentioned contributions:

- We propose a taxonomy of model transformation verification techniques and we use this taxonomy as a guide to review the state-of-the-art (Chapter 2). We have also previously conducted more extended surveys on model transformation verification techniques and intents [137, 138, 5, 4, 89, 3] to investigate how model transformation verification relates to other factors (e.g., the transformation intent, the property being proved, the transformation language used). This step
was intended to give us a higher-level view of existing verification approaches, and their strengths and limitations.

- Despite the existence of studies in industry adoption of MDD [41, 100, 149, 9] and to the best of our knowledge, no model transformation is reported to have migrated legacy models in the automotive industry. Thus, to test the practicality of using transformations for migrating industrial legacy models, we conduct an industrial case study where we report on the development of a migration model transformation (Chapter 3) and we report in more detail on the used tools and the challenges encountered in another study [141].

- We use the above-mentioned industrial case study to evaluate existing tools (Chapter 4, [136, 142]) and our property prover (Chapter 6, [140, 139]). Evaluating verification techniques using the same case study gives a common ground for the comparison of these techniques and has rarely been done in the literature.

- We extend and enhance a symbolic model transformation property prover for the DSLTrans model transformation language (Chapter 5, [140, 139]). We discuss the approach used by the new version of the property prover and the syntax and semantics of the property language. We show how the property language can be used to express commonly occurring properties, e.g., multiplicity invariants of different arities.

- We evaluate the property prover by using it to verify different property types of two transformations that vary in size and nature (Chapter 6, [140, 139]). The diversity of the two transformations and their properties gives more confidence in the wide applicability of our property prover.
For the above mentioned industrial case study, we demonstrate how our prover led to a two orders of magnitude improvement in verification time over an existing verification tool we used in Chapter 4 (comparison shown in Table 6.2 and in a former study [140]). We also discuss the strengths and limitations of our prover (Section 6.3).

1.3 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 proposes a taxonomy of the model transformation verification techniques investigated in the literature, and uses this taxonomy to guide the discussion of the related work. Chapter 3 describes an industrial model transformation that we developed and will be using as a case study to evaluate different model transformation verification tools in Chapter 4. Chapter 5 discusses our symbolic model transformation property prover and summarizes the graphical model transformation language which the prover is based on. Chapter 6 reports on two case studies conducted to evaluate our property prover, their verification and performance results, and the strengths and limitations of our prover. The thesis is concluded in Chapter 7 with a summary and possible future work.
Chapter 2

Related Work

We propose a taxonomy of the transformation verification techniques investigated in the literature, and we use this taxonomy to guide the discussion of the related work.

2.1 A Taxonomy of Model Transformation Verification Techniques

Figure 2.1 shows the taxonomy we propose for the state-of-the-art transformation verification techniques. Transformation verification techniques can be either static or dynamic verification techniques. Static verification techniques do not require executing the transformation of interest, unlike dynamic verification techniques.

Static verification techniques encompass Type I formal methods and Type II formal methods. Dynamic verification techniques encompass Type III formal methods, model checking, design space exploration, instrumentation, and testing. We explain static and dynamic verification techniques in more detail in Sections 2.2 and 2.3. We summarize the chapter in Section 2.4.
2.2 Static Verification Techniques

Static verification techniques verify transformations without executing them. Formal methods have been explored by many studies to statically verify transformation properties. Other static verification techniques developed originally for source code have also been adapted for transformations (e.g., using dependency graphs to ensure that rules are reachable [80]) but are rarely explored in the literature.

Formal methods are methods that use formalizations, e.g., Maude [39] (Appendix A), to formalize transformations and their input and output domains. Such formalizations are widely used since they have well-established techniques that can be used to verify properties of a transformation (e.g., termination) or a transformation’s output (e.g., type consistency). It has been argued that formal methods are computationally complex and hence are not scalable to complex transformations and input models [60].

We identify three types of formal methods used to verify transformations. We previously discussed the three types in [5, 137] and we summarize them in Table 2.1. Type I formal methods verify some property for all transformations when executed on any input model, i.e., they are transformation-independent and input-independent. Type II formal methods verify some property for a specific transformation when
executed on any input model, i.e., they are transformation-dependent and input-independent. Type III formal methods verify some property for a specific transformation when executed on one input, i.e., they are transformation-dependent and input-dependent. When a formal method is transformation-independent, it implies that no assumptions are made about the input. This explains the empty category in Table 2.1 representing formal methods that are transformation-independent and input-dependent.

Type I and Type II formal methods are discussed in this section since they are input-independent, and hence are classified as static verification techniques. Type III formal methods are classified as dynamic verification techniques (Section 2.3).

<table>
<thead>
<tr>
<th></th>
<th>Transformation-Independent</th>
<th>Transformation-Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input-Independent</td>
<td>Type I: [95], [49], [42], [148], [129], [28], [155], [112], [145], [11], [29], [86], [18], [50]</td>
<td>Type II: [6], [12], [58], [23], [90], [114], [61], [62], [106], [68], [36], [34], [35], [59], [32], [33], [17], [13]</td>
</tr>
<tr>
<td>Input-Dependent</td>
<td>-</td>
<td>Type III: [104], [16], [24], [110], [148], [129]</td>
</tr>
</tbody>
</table>

Table 2.1: Classification of formal methods used to verify model transformations.

### 2.2.1 Type I Formal Methods

Type I formal methods verify some property for any model transformation (i.e., transformation-independent) when run on any input model (i.e., input-independent).

**Verifying algebraic properties:** Algebraic properties include properties of transformations or their output that can be expressed using algebraic specifications. Stenzel *et al.* [145] used the theorem prover KIV to verify semantical properties of
operational QVT [8] transformations\(^1\), and type consistency and semantical properties of the transformation’s output\(^2\). The approach was demonstrated on a QVT transformation that maps class diagrams to Java Abstract Syntax Trees (JASTs). An algebraic formalization of a subset of QVT was implemented in KIV and was used to formalize the properties of interest. KIV was then used to verify the properties using the KIV-compatible format of the transformation. To verify properties of the transformation’s output, output JAST models were exported into a KIV-compatible format. KIV was then used to transform the JAST models in the KIV compatible format into an abstract syntax tree to verify properties of the Java code.

**Verifying semantics-preservation:** A transformation is *semantics-preserving* if the transformation’s output preserves the semantics of its input. Massoni et al. [95] presented an approach to develop a model refactoring that preserves structural semantics of class diagrams. The study defined an equivalence relation between UML class diagrams with OCL [158] constraints\(^3\) that can be used to identify the equivalence of class diagrams. The equivalence relation was based on defining equivalent elements that are common between the input and output domains, and a mapping function that defines the equivalence between elements that are not common between the input and output domains. This equivalence relation was used to define refactoring rules that are semantics-preserving. The class diagrams and the refactoring rules were then translated to an Alloy model for which semantics-preservation was verified with respect to the defined equivalence relation.

\(^1\)An example of a semantical property of a model-to-code transformation is that each UML class is transformed to a Java class.
\(^2\)An example of a semantical property of a transformation’s output code is that calling a setter then the corresponding getter returns the setter’s argument.
\(^3\)Object Constraint Language (OCL) [158] is a language originally developed to define constraints on UML models. The language was later adapted to define constraints on transformations, too.
Verifying Confluence: A confluent model transformation is a transformation that has a deterministic, unique output for every unique input.

Plump [112] verified confluence of hypergraph rewriting systems (Appendix A.1) using critical pair analysis. In critical pair analysis, all pairs of rules with a common left-hand side and which delete an element to be used by the other rule are computed. Such rule pairs are referred to as critical pairs since both rules can be executed but execution of one rule inhibits the other. If both rules in each computed critical pair reduce to a common hypergraph, then the transformation is confluent. Critical pair analysis needs only to be performed for rules with a nondeterministic execution order. A transformation without critical pairs or with a deterministic execution order is confluent. AGG [148, 129] is a tool that supports critical pair analysis and was used in other studies [49, 42] to verify confluence of graph rewriting systems. Critical pair analysis was used in other contexts too, e.g., parsing visual languages [30].

Assmann [11] guaranteed confluence of graph rewriting systems (GRSs, Appendix A.1) by ordering rules into a list of rule sets; the strata, based on the rule dependency graph. Rules in a stratum are forced to execute in an order that fulfills some conditions. Such conditions guarantee the generation of a unique output per stratum. Given that every stratum has a unique output and the list of all strata are computed in their stratification order, then the GRS has a unique output and is confluent.

Eramo et al. [50] developed an Eclipse plugin that assists developers in detecting (and refactoring) non confluent rule sets in design time in two steps. First, the transformation rules and the manipulated metamodels are translated into a logical representation. Then, the Answer Set Programming (ASP) solver can be used to deductively generate all the possible outputs that are consistent with the logical
2.2. STATIC VERIFICATION TECHNIQUES

representation corresponding to the transformation and its metamodels.

**Verifying Termination:** A terminating transformation stops executing after a finite number of steps. Although termination of a double pushout (DPO)-based, GRS is undecidable in general [113], some studies proposed sufficient termination criteria. Varró *et al.* [155] abstracted GRSs as Petri Nets where the initial marking of the Petri Net was determined by the input graph and tokens were passed based on the applicability of rules (represented as places) to the input graph. A matrix inequality was then solved to determine whether the Petri Net runs out of tokens in a finite number of steps, and hence whether the GRS terminates. Levendovszky *et al.* [86] proposed a criterion that states that if the right hand side (RHS) of a rule requires an extension to be mapped to the left hand side (LHS) of another rule, then the rule execution terminates. For cases where the RHS of a rule can be mapped to the LHS of another rule without extensions, then for infinite rule applications if each graph appears finitely many times, then the rule eventually terminates. The criterion assumed injective matches between rules and the host graph. Assmann [11] proposed two termination criteria. The first criterion assumes that a GRS only adds edges and that only one edge of a certain label is allowed between any two nodes. Thus, if a GRS adds edges while checking that the edges added do not already exist, then the GRS will terminate. The second criterion assumes that a GRS only deletes nodes and edges. Thus, elements are subtracted from the host graph until the GRS terminates.

Bottoni *et al.* [28] proposed a property that a function has to satisfy to be a valid, sufficient termination criterion of DPO-based, high level replacement units (HLRUs, Appendix A.1) without negative application conditions (NACs). The value of the function for the rule’s LHS must be greater than the value of the function for the
rule’s RHS. A HLRU terminates if any rule that is to be repeatedly applied has a valid termination criterion that it satisfies. The proposed property was extended to attributed, GRSs. One such criterion requires that the number of nodes and edges on the LHS of a rule is greater than the number of nodes and edges on the RHS of the rule. Bottoni and Parisi-Presicce [29] later improved the work in [28] and proposed a sufficient termination criterion for rules with NACs. The study measured the distance between the LHS and the NAC by measuring whether the number of matches increased or decreased after a rule application. This measurement was achieved by constructing a labeled transition system where states correspond to matches of a rule with all possible intermediate graphs between the LHS and the NAC of the rule. Transitions in the labeled transition system represented rule applications that moved the graph from one state to another state representing a graph instance closer to the NAC. On each transition, the number of rule matches were measured before and after applying a rule to determine whether the number of matches decreased with rule applications, and hence whether the rule terminates.

Ehrig et al. [49] proposed an approach to build terminating, DPO-based GRSs with NACs. The approach formulates a GRS as layers of rules with deletion and non-deletion layers. Each rule and each manipulated model element is assigned to a layer. Deletion layers contain rules that delete at least one element. Non-deletion layers contain rules that do not delete elements, cannot be applied twice to the same match, and cannot use a newly created item for the match. The layers must obey some layering conditions to terminate, e.g., the last creation of a node of a specific type should precede the first deletion of a node of the same type. The approach was demonstrated on a transformation from state charts to Petri Nets and it was verified
that the transformation terminates. However, the approach was found not applicable to transformations where rules are dependent on themselves [155].

de Lara and Taentzer [42] informally discussed how a transformation from process interaction models to timed transition Petri Nets (TTPNs, Appendix A.3) terminates and how layering conditions [49] can be used to guarantee termination. An intermediate metamodel composed of the source and target metamodels and auxiliary entities was used to represent the intermediate transformation outputs. The study also discussed how the transformation was found to produce output models that preserved syntactic consistency with respect to the target metamodel and behavioral equivalence with the input models. Syntactic consistency was proved by demonstrating how auxiliary elements of the intermediate metamodel were consistently removed by the transformation rules and how the rules created elements conforming to the target metamodel. Behavioral equivalence was argued by showing how example process interaction models behaved similar to their corresponding TTPNs.

Barroca et al. [18] proposed a graph-based transformation language, DSLTrans, that guarantees termination and confluence of transformations by construction. A DSLTrans transformation is composed of ordered layers where each layer contains rules that are executed in a non-deterministic order. Each rule has a match pattern (i.e., a pattern of the source metamodel) and an apply pattern (i.e., a pattern of the target metamodel). Given that the input model is acyclic, any DSLTrans transformation terminates since DSLTrans does not support recursion or loops, i.e., the limited expressiveness of DSLTrans helps guarantee termination. Additionally, any DSLTrans transformation is confluent since the only source of non-determinism (i.e., within a layer) is controlled by amalgamating the output of the rules in a layer using
2.2. STATIC VERIFICATION TECHNIQUES

graph union, which is commutative and hence, produces a deterministic output.

2.2.2 Type II Formal Methods

Type II formal methods verify some property for a specific transformation (i.e., transformation-dependent) when run on any input model (i.e., input-independent).

Verifying property-preservation: Giese et al. [58] used the theorem prover Isabelle/HOL to verify that a transformation specified using triple graph grammars (TGGs, Appendix A.2) in Fujaba\(^4\) preserves some safety property. Three manual steps were performed prior to verifying a transformation: (1) Isabelle/HOL algebraic representations for metamodels were derived from the TGG structures in Fujaba, (2) safety properties were then defined for such algebraic representations within Isabelle/HOL, and (3) TGG rules were formalized using Isabelle/HOL. Isabelle/HOL was then used to prove that the output of a TGG rule did not violate the equivalence of the input and output models with respect to the defined safety property. Becker et al. [23] verified safety requirements (expressed as forbidden graph patterns) for graph rewriting systems (GRSs) by checking if the backward application of each GRS rule to each forbidden pattern can result in a safe state and generating counter examples for these cases. The approach was implemented using graph manipulation data structures and using symbolic representations that can run on BDD engines. Both implementations were compared with model checking using GROOVE [124]. The results showed that GROOVE and the explicit implementation of GRSs were suitable for small examples, while the symbolic implementation scaled better to larger examples.

Poskitt et al. [114] represented a bidirectional transformation as two unidirectional

\(^4\)http://www.fujaba.de/
2.2. STATIC VERIFICATION TECHNIQUES

transformations ($S$ and $T$) in Epsilon that use EVL [76] to specify inter-model consistency constraints (denoted as $evl$). Verifying that $S$ and $T$ preserve consistency with respect to $evl$ was then done in two steps. First, the study translated $S$ and $T$ to GRSs and $evl$ to nested graph conditions. Second, the weakest precondition calculi for GRSs [63, 115] was leveraged and two specifications were verified using a theorem prover: $\{ evl \} S;T\{ evl \}$ and $\{ evl \} T;S \{ evl \}$. The two specifications mandate that if the input and output models initially preserve $evl$, then the output of executing $S$ and $T$ in any order preserves $evl$, too.

Verifying preservation of syntactic relations: Lúcio et al. [90] proposed a transformation checker for graph-based DSLTrans [18] transformations. Properties and rules of a DSLTrans transformation are expressed as (match, apply) patterns. The transformation checker builds the transformation’s state space where each state is a possible combination of the transformation rules, i.e., each state is a combination of (match, apply) patterns. Using the generated state space, the transformation checker verifies a property by checking that each state in the state space that has the property’s match pattern also has the property’s apply pattern. If at least one state has the property’s match pattern but does not have the property’s apply pattern, a counter example is produced. The transformation checker was used to verify a transformation and the study reported on the size of the generated state space.

Guerra et al. [61] proposed PaMoMo, a graphical language used to express visual contracts that specify input-output model relations. These properties can be compiled into OCL and injected into any OMG-based transformation implementation (e.g., ATL) for automated verification. Guerra et al. [62] later presented the PACO-Checker tool (PaMoMo Contract-Checker) which supports the visual specification of PaMoMo
contracts, their compilation into QVT-Relations, its chaining with the execution of the transformation of interest, and the visualization of the verification results.

Orejas and Wirsing [106] claimed that verifying triple graph grammars is difficult and hence, proposed generalizing triple graph grammars to triple algebras and graph constraints to algebraic patterns. The study then showed how the new formalization can be used in practice to verify properties expressed as input-output model relations.

**Verifying the typing of a transformation’s output:** Inaba et al. [68] used monadic second order logic (MSO) to verify the typing of a graph transformation’s output with respect to the output metamodel. MSO is first order logic with extensions to represent set quantification. The transformation of interest and the input metamodel were translated to MSO formulae and the conditions to be verified were translated to one MSO formula. The MONA solver was then used to prove whether or not the conditions will hold, and generated a counter example in the latter case.

**Verifying properties expressed as first-order logic:** Asztalos et al. [12] verified properties by formulating the transformation rules and the property of interest as assertions in first-order logic. Deduction rules were then used to deduce the property assertion from the transformation assertions. The proposed approach can be used to verify any property that is expressible in first-order logic, e.g., a rule application deletes all edges of a certain type. The approach was realized as a framework in the Visual Modeling and Transformation System (VMTS) since VMTS can automatically generate assertions representing the transformation rules. The framework was used to verify a property for a refactoring of business process models. The study claimed that their approach is extensible to different transformation frameworks and can be used to verify different properties. Disadvantages of the approach were also discussed,
2.2. STATIC VERIFICATION TECHNIQUES

e.g., inefficiency of the approach if complicated deduction rules were defined.

Cabot et al. [36] translated declarative QVT transformations and triple graph grammars into OCL invariants. These OCL invariants and the manipulated metamodels (collectively referred to as the transformation model) were transformed into a constraint satisfaction problem. The UMLtoCSP constraint solver was then used to either prove a property for the transformation model or disprove a property by generating a counter example. The study discussed properties that can be verified using the proposed approach (e.g., properties that are expressible in first-order logic).

Büttner et al. [35, 34] and Gogolla et al. [59] translated an ATL transformation and its metamodels into a transformation model and used model finders (e.g., USE Validator) to perform automatic and bounded verification (i.e., bounded by a scope) of OCL properties. Later, Büttner et al. [32] translated an ATL transformation and its metamodels into a first-order logic formalization, and used quantifier reasoning to perform automatic and unbounded verification of OCL properties with SMT solvers (e.g., Z3 Solver and Yices). Additional experiments were also published on-line [33].

Anastasakis et al. [6] proposed using UML2Alloy to translate a transformation and its metamodels to an Alloy model. The Alloy analyzer can then be used to verify the transformation in several ways. First, failure to produce a transformation instance using the Alloy analyzer can help identify inconsistencies in the transformation. Second, the Alloy analyzer can produce several transformation instances to explore different mappings between the source and target metamodels. Finally, the Alloy analyzer can perform assertion checking. The approach was demonstrated on a transformation for business process models and revealed an error in the transformation. The study discussed a few limitations of the approach, e.g., inapplicability to
2.3 Dynamic Verification Techniques

2.3.1 Type III Formal Methods

Type III formal methods verify some property for a specific transformation (i.e., transformation-dependent) when run on a specific input model (i.e., input-dependent).

Verifying structural correspondence between input and output models: Narayan and Karsai [104] verified structural correspondence between the input and non-declarative transformations. Baresi and Spoletini [17] also translated GRSs and their metamodels into an Alloy model. The Alloy analyzer was then used to analyze the applicability of a rule set to an initial graph, the graphs that are reachable from a specific number of rule applications on an initial graph, and whether a graph is reachable after a number of rule applications on the initial graph. The Alloy analyzer can also be used to prove if a property holds for at least one graph. The approach was demonstrated on a transformation and two case studies were conducted.

Asztalos et al. [13] implemented a verification tool in VMTS where transformations are expressed as graphical rules scheduled by a control flow graph. The tool assigns conditions to each edge in the control flow graph that are guaranteed to hold for the transformation at this edge. Conditions are assigned by analyzing individual rules to generate their strongest post-conditions and propagating these conditions using inference rules. Eventually, the final edge in the control flow graph is assigned the strongest post-condition $p_{\text{final}}$ of the transformation. A property $p$ is then verified by evaluating $p_{\text{final}} \rightarrow p$. One limitation of the tool is that property verification may be undecidable due to (1) the lack of the necessary inference rules or (2) the need to collectively analyze the control flow graph instead of analyzing rules separately.
output models of graph rewriting systems specified in GReAT [2] in three steps. First, a composite metamodel that contains the source and target metamodels and structural correspondence rules that manipulate elements of the two metamodels were defined in GReAT. Second, the transformation of interest was extended with crosslinks between input model elements and their corresponding output model elements. Third, the crosslinks were used to evaluate the structural correspondence rules for a specific pair of input and output models. The approach was demonstrated on a transformation from UML activity diagrams to communicating sequential process models. AGG [148, 129] verifies graph constraints for a specific pair of input and output models of a graph transformation. Specifically, AGG checks that if premise holds before a rule application then a conclusion also holds after the rule application. AGG does not check all transformation executions; only the first found execution is verified. AGG performs other types of analysis, e.g., critical pair analysis and graph parsing.

Verifying semantics-preservation: Baar and Marković [16] proposed an approach to prove that a model refactoring is semantics-preserving. A transformation that refactors a UML class diagram with OCL constraints and a set of conforming object diagrams is said to preserve static semantics if evaluating the unrefactored constraints on the unrefactored object diagrams produces the same results as evaluating the refactored constraints on the refactored object diagrams. The refactoring rules and the evaluation of OCL constraints were formalized as graph rewriting rules. The approach was used to prove that a model refactoring preserved static semantics.

Verifying preservation of type consistency and multi-view consistency: A transformation preserves type consistency if it generates outputs that are well-formed with respect to the target metamodel and its constraints. A model preserves
multi-view consistency if multiple views of the model do not contradict each other.

Becker et al. [24] proposed an approach to verify that refactoring a metamodel of a modelling language preserved type consistency with respect to well-formedness constraints of the modeling language that cannot be specified as a (conditional) forbidden patterns, e.g., two methods in the same class cannot have the same signature. Model refactoring was formalized as a graph rewriting system. The source metamodel was extended with predicate structures and indirect well formedness constraints were specified as graph constraints that manipulate the predicate structures. Maintenance rules were evaluated after every refactoring rule to add predicates to the model if the model has a forbidden pattern. If the refactored model overlapped with a forbidden pattern, a counter example was generated. Using the approach, two refactorings of the java language metamodel were proven to be consistency preserving and two bugs in a refactoring used by Eclipse were discovered, one of which was only recently fixed.

Paige et al. [110] used PVS [108] and Eiffel [99] to perform model conformance checking and multi-view consistency checking (MVCC) between different diagrams of a BON [157] model. BON is a modeling language that does not have formally defined semantics. PVS is a theorem prover and Eiffel is an object-oriented programming language that supports contracts. To represent the BON metamodel in PVS, BON metamodel entities were represented as theory constructs and the metamodel constraints were represented as axioms that manipulate the theory constructs. To prove model conformance in PVS, a BON model was encoded as PVS expressions and PVS was used to prove that the encoded BON model satisfies the axioms encoding the BON metamodel constraints. To perform MVCC between diagrams in PVS, the PVS theory was extended with lemmas to enable representing different views of a system
2.3. DYNAMIC VERIFICATION TECHNIQUES

with constraints. To perform MVCC between contracts of diagrams, preconditions of successive routines were composed into one axiom. Then, a BON model was encoded as a PVS conjecture that satisfied the axiom. On the other hand, representing the BON metamodel in Eiffel was straightforward since all constructs in BON have equivalent constructs in Eiffel. To prove model conformance in Eiffel, a BON model was encoded as an object, and the metamodel rules were executed on the object to check if the model conforms to the metamodel. MVCC between diagrams in Eiffel was performed in a similar manner to model conformance checking. MVCC between contracts of diagrams was performed in Eiffel by generating unit tests from the encoding of dynamic diagrams, and generating Eiffel code from the encoding of class diagrams, and running the unit tests against the Eiffel code. The study compared PVS and Eiffel qualitatively only.

2.3.2 Model Checking

Many studies verified model transformations specified using some formalization (Appendix A) by model checking the state space of the transformation.

Model checking Maude specifications: Several studies proposed translating graph-based transformations to Maude to facilitate verification. Boronat et al. [27] used Maude (Appendix A.5) to formalize a transformation’s source metamodel as a membership equational theory and to formalize models as terms of the membership equational theory corresponding to the source metamodel. Accordingly, a transformation was formalized as a rewrite theory that operates on terms of a membership equational theory. Maude was then used to perform three types of analysis: simulation of the transformations, reachability analysis to prove invariant satisfaction,
and analysis of linear temporal logic (LTL) [39] properties. The approach was implemented as an Eclipse plugin, MOMENT2. An exemplar transformation was verified using MOMENT2, which uncovered a violated LTL property. Similarly, Rivera et al. [127] integrated the Maude code generator with AToM3 as a visual front-end to specify graph rewriting systems (GRSs) and automatically translated a GRS and its manipulated graphs into Maude for verification. Reachability analysis results were transformed back to the visual language of AToM3. The study demonstrated how the approach helped in revealing properties that were not satisfied by a transformation.

Troya and Vallecillo [152] translated textual ATL transformations into a rewriting theory in Maude, and used Maude to perform transformation simulation, reachability analysis, and model checking (i.e., verifying the trace model of a specific execution).

**Model checking graph rewriting systems (GRSs):** Rensink [123] model checked graph transition systems (Appendix A.1) for temporal properties such as logic expressions on edge labels and node set expressions. The study discussed the semantics of the temporal expressions and their evaluation for a graph. Using the proposed semantics, temporal properties can be easily verified for the states in a graph transition system. Later, Rensink [125] extended his work in [123] to formally define and evaluate graph-based, linear temporal logic expressions for graphs. The proposed approach was implemented as a tool called GROOVE [124]. GROOVE supports stepwise, manual execution of rules and automatic generation of a graph transition system (i.e., the state space). Model checking the graph transition system for temporal properties was left for future work. GROOVE was demonstrated on a transformation and the results were discussed in terms of the size of the generated state space.
2.3. DYNAMIC VERIFICATION TECHNIQUES

Rensink et al. [126] compared CheckVML and GROOVE for model checking GRSs. CheckVML transforms a GRS and an initial graph to a Promela model, where graphs are encoded as fixed state vectors and GRS rules are encoded as guarded commands that modify the state vectors. The Promela model is then verified using the SPIN model checker. On the other hand, GROOVE was used to build the state space of a GRS for model checking. The two approaches were evaluated on three transformations with respect to the size of the generated state space, the memory usage, and the execution time. The study concluded that GROOVE is better for problems where processes and resources are not distinguished from one another (i.e., no concurrency).

Arendt et al. [10] proposed Henshin, a tool that generates a state space of possible graph transformation executions for a specific input and provides an extension point for model checkers. Narayanan and Karsai [103] used the GReAT framework [2] to verify bisimilarity between the input and output models of a GRS with respect to reachability. The approach was demonstrated on a GRS that transforms state charts to Extended Hybrid Automata (EHA) models. EHA provides formal semantics for state charts and hence are more appropriate to use for verification. An EHA model is bisimilar to a state chart if a reachable state configuration in a state chart has an equivalent reachable state configuration in an EHA model and vice versa. GReAT maintains cross-links between input and output model elements that can be used to prove bisimilarity. For every transition in a state chart and its equivalent transition in its corresponding EHA model, the minimal source state configuration is computed for the transition in both models. Equivalence between the start and end state configurations of each pair of equivalent transitions implies that the state chart and the EHA models are bisimilar. If the models are bisimilar, then the EHA model
can be transformed to a Promela model and analyzed for reachability using the SPIN model checker. The trace generated by SPIN to prove reachability corresponds to a transition sequence in an EHA model. Using the cross links in GReAT, the transition sequence in the EHA model can be traced back to the transition sequence in the state chart. The proposed approach uses both Type III formal methods (Section 2.3.1) to verify bisimilarity and model checking to verify reachability.

**Model checking Petri Nets:** König and Kozioura [78] proposed Augur2, a tool that approximates GRSs with Petri Nets and then verifies structural properties of the resultant Petri Nets for all reachable markings. The GRS and the initial graph are used to generate the state space of graphs which is mapped to a state space of Petri Net markings. The user can specify a property as a graph pattern which is mapped by Augur2 to an equivalent Petri Net marking. Augur2 either proves that the property holds or produces a counter example which is an execution of the Petri Net producing a marking that represents a graph violating the property being verified.

**Model checking programs generated by model-to-code generators:** Ab rahim and Whittle [1] verified the conformance of UML State Machine-to-Java code generators with respect to the semantics of UML state machines using model checking. Assertions that capture the semantics of state machines were defined and compiled into a single verification component. Annotations that refer to the verification component were also appended within the generated code. Using the annotations, Java Path Finder (JPF) was used to model check the conformance of the generated code to the source language semantics by checking if any assertion was violated. Two case studies were conducted on the state machine-to-Java code generators of IBM Rhapsody [67] and Visual Paradigm [69]. The results revealed that the tools did not fully
conform to the semantics of UML state machines.

2.3.3 Design Space Exploration (DSE)

Design Space Exploration (DSE) verifies transformation outputs that meet some design constraints or that achieve acceptable values for non-functional metrics [65].

Hegedus et al. [65] performed guided DSE of graph rewriting systems (GRSs) given some global constraints on all states and goals on solution states. Using predefined selection criteria to prioritize promising paths and cut off criteria to prune unpromising paths, DSE is executed in a series of steps. For the current state, all cutoff and selection criteria are evaluated and the applicable rules are identified. If a cutoff criterion holds or if there are no applicable rules, then the state is marked as a dead end. Otherwise, the next applicable rule is selected (based on the selection criteria) and applied to the current state to generate a new state. If the new state is a solution as specified by the goals, then the solution trajectory (i.e., applied rules and final state) is saved and the next applicable rule is applied to a new state. However, if the new state does not satisfy the global constraints then search continues from the previous state. If the new state is not a solution but satisfies the global constraints, search continues from the same state. DSE stops if a predefined number of solutions are found or if the state space was searched exhaustively. The approach was found to generate an optimal solution trajectory earlier than the depth first DSE.

Drago et al. [47] proposed QVT-Rational to perform DSE of transformations in three steps. First, an expert specifies the manipulated metamodels, the quality metrics of interest, a quality prediction tool chain, and a quality-driven transformation.\(^5\)

\(^5\)Quality-driven transformations are implemented in transformation languages with constructs that support representing non-functional transformation attributes.
The expert also binds quality metrics to the different transformation mappings. Second, a designer specifies the input model and the requirements for the quality metrics. Third, the designer runs the framework to get viable outputs and their quality predictions. The framework was demonstrated on two transformations and was found to scale well for inputs of varying sizes. The study also discussed some disadvantages of the framework, e.g., dependence of the quality of the viable outputs on the experience of the domain expert who bound different mappings to different quality metrics.

Schätz et al. [131] formalized models as Prolog terms, transformation rules as Prolog predicates, and the transformation state space as (pre-model, post-model) relations. The formalization of the rules was then interpreted by Prolog as a non-confluent transformation. The approach was demonstrated on a transformation and optimizations were implemented to decrease the runtime and memory usage of the approach.

2.3.4 Instrumentation

Instrumentation involves adding *instrumentation code* to the transformation to debug its inner workings. Instrumentation of transformations was rarely investigated in the literature for the purpose of verification. Dhoolia et al. [45] dynamically tagged model-to-text transformations to debug faulty input models in three steps. First, the user specifies *markers* in the output to mark a faulty output substring. Second, the transformation is executed in the debug mode where the transformation associates a tag with each input model entity and propagates the tags to the corresponding output substrings. This execution generates a log file with the faulty output, tags, and preset fault marker. Finally, the user traverses the log file to locate the faults.
in the input. A case study was conducted on six transformations. For all faults, the fault search space was either significantly decreased or precisely identified. However, the run-time and the size of the instrumented transformation increased in some cases. The approach can be extended to debug transformation faults. The output tags can additionally save the statement in the transformation that propagated the tags. The tags can then be used to identify which faulty transformation statements produced the fault in the output.

2.3.5 Testing

Testing executes a transformation on input models and validates that the actual output matches the expected output [64]. Several studies have discussed the challenges facing transformation testing [19, 20, 55, 81, 88] and we previously summarized these challenges [137]. Despite these challenges and despite the fact that testing does not fully verify the correctness of a transformation, it has been gaining increasing interest for many reasons. The major advantage of testing is its usefulness in uncovering bugs while maintaining a low computational complexity [60]. Other advantages include the ease of performing testing activities, the feasibility of testing the transformation in its target environment, and the ease of automating most of the testing activities [88].

In this section, we differentiate between a transformation’s implementation and specification. A transformation implementation consists of the rules that map between the source and target metamodels.\(^6\) By contrast, a transformation specification includes the source and target metamodels (and their constraints), and the transformation contracts. A contract is composed of three sets of constraints [38]:

\(^6\)We use the notions of a model transformation and a model transformation implementation interchangeably.
(1) constraints on input models, (2) constraints on output models, and (3) constraints on input-output model relations.

Several studies have proposed taxonomies of transformation contracts. Baudry et al. [19] define three levels of contracts: transformation contracts, sub-transformation contracts, and output contracts. (Sub-)transformation contracts include pre-/post-conditions of (sub-)transformations and their invariants. Output contracts are expected properties of output models. Mottu et al. [102] categorized contracts as either syntactic or semantic. Syntactic contracts ensure that the transformation can run without errors. Semantic contracts are context-dependent and can be subdivided into preconditions on input models, postconditions on output models, and postconditions linking input and output models.

In what follows, we propose four transformation testing phases. Then, we survey the state of the art related to each phase.

An Overview of Model Transformation Testing Phases

We break down the transformation testing process into four phases, inspired by those defined by Baudry et al. [20] with minor changes. The first phase, test case generation, involves generating a test suite or a set of test models conforming to the source metamodel. Adequacy criteria are used to generate an efficient test suite to test the transformation. The percentage of adequacy criteria satisfied by a test suite is referred to as the coverage achieved by the test suite [96] (Eqn. 2.1).

$$\text{Coverage} = \frac{|\text{AdequacyCriteriaSatisfiedByTestSuite}|}{|\text{AdequacyCriteria}|} \times 100\% \quad (2.1)$$

The second phase is assessing the test suite. A test suite that has a positive assessment is more likely to expose faults in a transformation. A test suite that has a negative assessment can be improved by adding relevant models to the test suite.
The third phase is building the oracle function. The oracle function is the function that compares the actual output of a transformation with the expected output to evaluate the correctness of the transformation [55].

The fourth phase is running the transformation on the test suite and evaluating the actual outputs using the oracle function. For each model in the test suite, if the oracle function detects a discrepancy between its corresponding actual and expected outputs, then the tester can examine the transformation and fix bugs accordingly.

We survey studies related to the first three phases. The fourth phase is a straightforward process given that the test suite and the oracle function are built correctly.

**Phase I: Test Case Generation**

Test case generation involves defining test adequacy criteria and building a test suite that achieves coverage of the adequacy criteria. Defining test adequacy criteria, and hence test case generation, can follow a black-box, grey-box, or white-box approach. A black-box approach assumes that the transformation implementation is not available and builds a test suite based on the transformation specification (i.e., source metamodel or contracts). A grey-box approach assumes that the transformation implementation is partially available and builds a test suite based on the accessible parts of the transformation implementation [64]. A white-box approach assumes that the full transformation implementation is available and builds a test suite based on the transformation implementation. We discuss criteria proposed for black-/white-box test case generation in more detail. We do not discuss grey-box test case generation, since it has been rarely investigated in the literature. Moreover, grey-box test case generation can use the same approaches as those proposed for white-box test case
2.3. DYNAMIC VERIFICATION TECHNIQUES

generation but only on the accessible parts of the transformation implementation.

Black-Box Test Case Generation Based on Metamodel Coverage

Different adequacy criteria have been proposed to achieve coverage of the different source metamodels of transformations\textsuperscript{7}, e.g., if a transformation manipulates class diagrams, then adequacy criteria for class diagrams can be leveraged for black-box testing.

McQuillan and Power [96] surveyed the black-box adequacy criteria proposed in the literature for one structural model and five behavioral models. The study reviewed how the criteria were evaluated and concluded that little work has been done on evaluating the effectiveness of the criteria in detecting faults and on comparing the criteria in terms of the coverage they provide. We summarize adequacy criteria proposed for class diagrams since they are the only structural models with criteria proposed in the literature. Due to space limitations, we summarize the adequacy criteria for only two behavioral models (interaction diagrams and state charts).

**Adequacy Criteria for Class Diagrams:** Three criteria have been investigated for class diagrams [7, 55, 57]: the association-end multiplicity (AEM) criterion, the generalization (GN) criterion, and the class attribute (CA) criterion. The AEM criterion requires that each representative multiplicity-pair of two association ends gets instantiated in the test suite. The GN criterion requires that each subclass gets instantiated. The CA criterion requires that each representative class attribute value gets instantiated.

In the AEM and CA criteria, representative values are used since the possible values of multiplicities and attributes can be infinite. Representative values are created

\textsuperscript{7}We survey black-box adequacy criteria for testing models and transformations, since in both cases, criteria are dependent on the input metamodel only.
using *partition analysis* [107] where multiplicity and attribute values are *partitioned* into mutually exclusive ranges of values. A representative value from each range must be covered in the test suite. To build partitions, default partitions can be automatically generated or knowledge-based partitions can be generated by the tester [53].

Some studies [55, 57] propose the notion of a *coverage item*, which is a constraint on the test suite that requires certain combinations of objects, representative CA values, and AEM values to be instantiated in the test suite. A test adequacy criterion can then be defined for each coverage item. Fleurey et al. [53] also combined classes, representative CA values, and representative AEM values into coverage items. A coverage item for an object was referred to as an *object fragment*. A coverage item for a model was referred to as a *model fragment* and is composed of several object fragments. The study then proposed different adequacy criteria specifying different ways of combining object fragments into a model fragment. A tool was built to implement the proposed criteria and to guide the tester by generating the required model fragments and to point out model fragments that were missing in the test suite. The tool was found to suggest model fragments that are not feasible, e.g., suggesting a model fragment with zero transitions and one transition in an input state machine.

**Adequacy Criteria for Interaction Diagrams:** Seven adequacy criteria have been investigated for interaction diagrams [7, 57, 159]: each message on link (EML), all message paths (AMP), collection coverage (Coll), condition coverage (Cond), full predicate coverage (FP), transition coverage, and all content-dependency relationships coverage. The EML criterion requires that each message on a link connecting two objects gets instantiated in the test suite. The AMP criterion requires that each possible sequence of messages gets instantiated. The Coll criterion requires
each interaction with collection objects of representative sizes gets instantiated. The Cond criterion requires that each condition gets instantiated with both true and false. The FP coverage criterion requires that each clause in every condition gets instantiated with both true and false such that the value of the condition will always be the same as the value of the clause being tested. The transition coverage criterion requires that each transition type gets instantiated. The all content-dependency relationships coverage criterion is based on extracting data-dependencies between system components and requires that each identified dependency gets instantiated.

**Adequacy Criteria for Statecharts:** Six adequacy criteria have been investigated for statecharts [105, 159, 64, 159]: full predicate (FP) coverage, all content-dependency relationships coverage, transition coverage, transition pair coverage, complete sequence coverage, and all-configurations-transitions coverage for statecharts with parallelism. The FP coverage criterion, the all content-dependency relationships coverage criterion, and the transition coverage criterion are similar to their equivalents for interaction diagrams. The transition pair coverage criterion requires that each pair of adjacent transitions gets instantiated in the test suite. The complete sequence coverage criterion requires that each complete sequence of transitions that makes full use of the system gets instantiated. The all-configurations-transitions coverage criterion for statecharts with parallelism requires that all transitions between all state configurations in the reachability tree of a state chart get instantiated. Similar to class diagrams, coverage items can be created for interaction diagrams and statecharts and test adequacy criteria can be defined accordingly.

**Black-Box Test Case Generation Based on Contract Coverage** Adequacy criteria have been proposed in the literature to achieve coverage of transformation
contracts. Fleurey et al. [55] proposed constructing an effective metamodel composed of only the source metamodel elements that are used in a transformation’s pre-/post-conditions. The values of attributes and multiplicities in the effective metamodel can then be partitioned, and the partitions can be used to generate coverage items and adequacy criteria. No case study was conducted to evaluate the proposed approach.

Bauer et al. [22] propose a combined specification-based coverage approach for testing a transformation chain, where contract-based and metamodel-based criteria were generated from the constituent transformations. Contract-based criteria required the execution of each contract by the test suite. Traditional metamodel-based criteria, e.g., the AEM criterion, were used. Using the generated criteria and an initial test suite, a footprint was generated for each test model. A footprint is a vector of the number of times a test model covers each criterion. The quality of the test suite was then measured using the footprints of all the test models to assess the (un)covered criteria and the redundant test models. A case study was conducted on a transformation chain with 188 test models. Several adequacy criteria were found to be unsatisfied and adding test models to cover these criteria revealed faults in the transformation chain. Moreover, 19 redundant test models were identified and removed.

Bauer and Küster [21] investigated the relationship between specification-based (black-box) adequacy criteria used in [22] and code-based (white-box) adequacy criteria derived from the control flow graph of a transformation chain. Such a relation can be useful in many ways. First, the relation can be used to determine parts of the specification that are relevant to a code block and vice versa. Second, the relation can be used to identify code and specification relevant to a test model to facilitate debugging the transformation for failing test models. Third, the relation can be used
2.3. DYNAMIC VERIFICATION TECHNIQUES

to determine how closely related the two types of criteria are and hence how closely the code reflects the specification. The proposed approach was evaluated and it was found that the coverage achieved for the code-based and specification-based criteria are linearly correlated. Thus, properties of code blocks can be deduced from their related specifications without having to manually analyze the code.

**White-Box Test Case Generation**  Different adequacy criteria have been proposed in the literature to achieve coverage of a transformation implementation. Fleurey *et al.* [55] proposed using a static type checker to build an *effective* metamodel composed of the source metamodel elements referenced in the transformation implementation. Attributes and multiplicities that constitute the effective metamodel can then be partitioned and the defined partitions can be used to generate coverage items and adequacy criteria. No case study was conducted to evaluate the proposed approach.

Küster and Abd-El-Razik [81] proposed three white-box approaches to test transformations specified as conceptual rules and built using IBM WebSphere Business Modeler. The first approach was based on transforming a conceptual rule into a source metamodel template, from which model instances can be created automatically. To create a source metamodel template from a rule, abstract elements in conceptual rules must be parameterized. Thus, several templates were generated from each rule to ensure source metamodel coverage per rule. The second approach experiments with output models with constraints. For each constraint on an output model element, a test model that affects the constraint was generated. The third approach generated critical input models that contain overlapping match patterns of rule pairs to test if errors can occur due to the interplay of rules. The study concluded that the third approach revealed fewer errors than the first two approaches.
McQuillan and Power [97] assessed the coverage of ATL rules by profiling their operation during execution. ATL has two features which allow it to support such profiling. First, compiled ATL rules are stored in XML files and are executed on top of a special purpose virtual machine. Second, ATL prints out a log file of the executed instructions. To assess rule coverage of ATL transformations, a two phase-approach was proposed. First, the XML file resulting from compilation of the transformation is processed to extract the available rules and helpers. Second, the transformation was executed using the available test suite. The resulting log file was processed to find out how many rules and helpers extracted in the first phase were covered according to three white-box criteria: rule coverage, instruction coverage, and decision coverage.

Lämmel [82] proposed a criterion for grammar testing that can be used for transformation testing since using a grammar to transform a language is similar to using a transformation to transform models. The study proposed a criterion that requires each rule to be triggered in every possible context. For example, if the output of rule $r_1$ can trigger either rule $r_2$ or rule $r_3$, then there must be one test model that triggers $r_1$ then $r_2$, and another model that triggers $r_1$ then $r_3$. No case study was conducted to evaluate the efficiency of the criterion in detecting faults.

**Phase II: Test Suite Assessment**

Many studies used a test suite’s coverage with respect to some adequacy criteria (discussed in previous sections) to assess the quality of test suites [7, 55, 57, 53, 159, 105, 64, 96, 22, 21, 81, 97]. Others used mutation analysis instead [102, 83, 96, 101, 105].

Mutation analysis [101] is a technique used to evaluate the sensitivity of a test
suite to transformation faults. In mutation analysis, *mutation operators* are used to inject faults in the original transformation and generate *mutants*. The injected faults are determined using *fault models* which capture developers’ errors when building transformations. The mutants and the transformation are then executed using the test suite under assessment. For each mutant, if one test model produces different results for the transformation and the mutant, then the mutant is *killed*. The mutant stays *alive* if no test model detects the injected fault. A mutant that can not be killed by any test model is an *equivalent mutant* and has to be discarded. A *mutation score* is computed to evaluate the test suite, as shown in Eqn. (2.2).

\[
\text{MutationScore} = \frac{|\text{KilledMutants}|}{|\text{Mutants}| - |\text{EquivalentMutants}|}
\]

Mottu *et al.* [101] proposed mutation operators that model semantic faults which are normally not detected when programming, compiling or executing a transformation. Four basic transformation operations were identified: input model navigation, filtering of the navigation result, output model creation or input model modification. The study then proposed mutation operators related to the four operations. Using the proposed mutation operators, mutants were generated for a Java transformation and were compared with the mutants generated using MuJava. MuJava uses mutation operators that exist in any programming language and are not dedicated to MDD. MuJava generated almost double the number of mutants generated from the proposed operators, with more mutants being not viable, i.e., detected at compile-/run-time.

Dinh-Trong *et al.* [46] discussed three fault models for UML models that can be used for transformations: design-metric related faults, faults detectable without execution, and behavior-related faults. Design metric related faults result in undesirable values for design metrics that can imply problems in non-functional properties. Faults
2.3. DYNAMIC VERIFICATION TECHNIQUES

Detectable without execution result from syntactic errors and are easily killed by MDD environments. Behavior-related faults result from an incorrect transformation specification that is syntactically correct. Several mutation operators were discussed and classified according to the proposed fault models.

**Phase III: Building the Oracle Function**

An oracle function compares the actual output with the expected output to validate a transformation [55]. If the expected output models are available, then the oracle function is a model comparison or a model differencing task [87, 77, 88]. If the expected output models are not available, then the oracle function validates the transformation’s output with respect to output specifications or contracts [38, 37, 102, 60, 83].

**Model Comparison as Oracle Functions:** Model comparison or differencing has been identified as a major task in transformation testing [77]. Lin et al. [88] proposed a testing framework where test cases are automatically generated, model comparison is used as the testing oracle function, and the comparison results are visually presented. A case study was conducted and it was shown that the scalability of the approach is limited due to the use of graph matching. Since model comparison is a stand-alone topic with many dimensions [87], we do not discuss it any further.

**Contracts as Oracle Functions:** Contracts specify expected properties of the transformation’s output, and can be used as oracle functions. Many languages for defining transformation contracts have been proposed, e.g., Java Modeling Language (JML) [84] can be used to define contracts for Java transformations. However, OCL [158] has been used in many studies for specifying transformation contracts.
Cariou et al. [38] discussed two approaches to specify OCL constraints on input-output model relationships. In the first approach, OCL expressions that manipulate elements of a single metamodel were specified in the transformation’s postcondition. In this approach, the mapping between input and output model elements is implicit, i.e., input elements that are not manipulated in the postcondition will automatically be maintained in the output. Moreover, OCL expressions are simple due to the use of one metamodel for both the source and target. On the other hand, a disadvantage of this approach is that it can only be used when the source and target metamodels are the same since the transformation must be owned by a classifier of one metamodel. Thus, for transformations manipulating different metamodels, a common metamodel needs to be defined. Finding a common metamodel is not always easy; the metamodels may have contradicting constraints. Further, the classifier must be carefully chosen to enable all elements in the OCL expressions to be referenced. The first approach was demonstrated on two transformations [37].

In the second approach, OCL expressions that manipulate models as packages were specified in the postcondition. The second approach can be used to define contracts for transformations that manipulate different metamodels. Disadvantages of the second approach include the need for an OCL extension to define explicit mappings between input and output model elements. Further, OCL expressions can be verbose due to the use of two metamodels. The study applied the second approach to a transformation to demonstrate the definition of an OCL extension and the definition of the contracts.

Gogolla and Vallecillo [60] proposed a framework for testing transformations based
on a generalized type of contracts called tracts. A tract defines a set of OCL constraints (source tract constraints, target tract constraints, source-target tract constraints) and a tract test suite. Source tract constraints are constraints on input models; target tract constraints are constraints on output models that must be satisfied together with the target metamodel constraints; source-target tract constraints are constraints on input-output model relationships; the tract test suite is a test suite built to satisfy the source tract constraints and the source metamodel constraints. The context of the tract constraints was a tract class that contained functions and attributes used to specify the tract constraints. A framework was implemented and was used to verify a transformation. The paper discussed the advantages of using tracts in testing. However, no case study was conducted to evaluate the framework.

Improving Transformation Contracts: Some studies focused on the importance of contracts and the need to improve them. Two approaches were proposed to improve contracts [102, 83]: mutation analysis [102] and mathematical modeling [83]. Both approaches aimed to improve two transformation metrics that reflect the effectiveness of its contracts: vigilance and diagnosability. Vigilance is the probability that the contracts dynamically detect errors [102, 83]. Diagnosability is the effort needed to locate a fault once it has been detected by a contract [83].

Mottu et al. [102] improved a transformation’s vigilance by improving the consistency between the transformation’s test suite, implementation, and contracts in three steps. First, an initial test suite was analyzed repeatedly using mutation analysis until an acceptable mutation score was achieved. Second, the optimized test suite was used to test the transformation and fix errors. If the final, fixed transformation differs significantly from the original one, mutation analysis was repeated since
different mutants can be generated. Finally, the contracts’ accuracy was evaluated using mutation analysis. If a mutant was killed by a test model but was not killed by any contract, then a contract had to be added. The approach was evaluated on a transformation and the contracts’ mutation score was improved to detect up to 90% of the mutants detected by the test suite.

Le Traon et al. [83] used contracts to improve a system’s vigilance and diagnosability using mathematical modeling. Although the study focused on systems captured as models with OCL constraints, the approach can be used for transformations with contracts. A system’s vigilance was expressed as a function of the isolated and local vigilance of its components and the probability that this specific component causes a failure. A system’s diagnosability was expressed as a function of two attributes: the probability that a faulty statement in a set of statements bounded by two consecutive contracts is detected by any contract that comes after the fault and the diagnosis scope. Three case studies were conducted and it was proven that a system’s vigilance and diagnosability improved significantly with the addition of contracts.

2.4 Summary

In this chapter, we propose a taxonomy of transformation verification techniques that have been proposed in the literature and we use this taxonomy to guide our survey of the related work. We classify verification techniques as either static or dynamic techniques. Static techniques include Type I and Type II formal methods. Dynamic techniques include Type III formal methods, model checking, DSE, instrumentation, and testing. For each class of verification techniques, we survey the related approaches in the literature.
We previously provided a more comprehensive survey of the state-of-the-art in transformation verification [5, 137, 3] and testing [138]. In the transformation verification surveys [5, 137, 3], we identified and investigated three dimensions of transformation verification: the verification approach used, the verified properties, and the transformation language under verification. Besides surveying verification techniques, the studies [5, 137, 3] can be viewed as catalogues that aim to assist transformation developers in many ways, e.g., choosing a suitable technique to verify a certain property for a transformation implemented in a specific language. We have also previously conducted a collaborative study that proposed a catalogue for transformation intents [4, 89]. For each intent, we identified aspects of interest for transformations belonging to that intent, e.g., relevant properties and verification techniques used to prove these properties. Both studies classified previous research using the proposed catalogue and with reference to the above-mentioned aspects.
Chapter 3

The GM-to-AUTOSAR Model Transformation

In this chapter, we present a model transformation that we have previously developed in an industrial context for General Motors (GM). Basically, the transformation migrates legacy GM models into their equivalent AUTOSAR models. We use our GM-to-AUTOSAR model transformation later in Chapter 4 as a case study to experiment with a black-box testing tool and an automated formal verification tool. Experimentation with state-of-the-art tools is intended to give us a better understanding of the limitations of current tools which we aim to address in our tool (discussed in Chapter 5).

In Section 3.1, we review the development process which provides a context for our GM-to-AUTOSAR model transformation. Next, we describe the model transformation problem, including the source GM metamodel and the target AUTOSAR metamodel manipulated by our transformation in Section 3.2. Then, we discuss the design, development, and implementation of our GM-to-AUTOSAR model transformation in Section 3.3. Finally, we summarize this chapter in Section 3.4.
3.1 Context: Vehicle Control Software, Models, and Transformations

For Vehicle Control Software (VCS) development, the relevant process artifacts include design stages and activities, and the input and output models of each stage.

Typical VCS Development Process and Models  The VCS development process is typically described as a V-diagram [122], shown in Fig. 3.1. In this process, the stages on the left-hand side of the V-diagram are activities related to design and implementation, and the stages of the right-hand are activities related to integration and validation. The design starts from system requirements models, which are decomposed into hardware and software subsystem requirements models. The subsystem requirements models then are assigned to engineering groups or external organizations for refinement into design models and then implemented by hardware and software components. These implemented components are integrated into Electronic Control Units (ECUs), configured for a designated vehicle product. The components are then tested and validated at various levels (as shown on the right-hand side of the V-diagram) against their corresponding models on the same level on the left-hand side of the V-diagram.

Different types of models are used and generated in the process, including control models and hardware architecture models. The models use different formalisms: control models use differential equations and timing-variation functions; and hardware architecture models uses annotated block diagrams. Selected modeling tools (e.g., Simulink, Rhapsody) and languages (e.g., UML, AADL) are used for modeling.

Model Transformation Types in the VCS Development Process  Given the model types used in the VCS development process, the transformations manipulating
these models can be classified into two categories:

- **Horizontal transformations.** Horizontal transformations manipulate models at the same abstraction level [98]. Examples include the transformation of a state machine in Matlab Stateflow into a UML 2 state machine. Such transformations are normally used to verify integration when subsystems/components are composed to realize a system function. The modeling languages for the source and target models may have different syntax, but must share similar, or overlap in, semantics.

- **Vertical transformations.** Vertical transformations manipulate models at different abstraction levels [98]. Examples include generation of a deployment model from software and hardware architecture models. Vertical transformations are usually more complex than horizontal transformations due to the different semantics of the source and target models.
3.2. The Model Transformation Problem Description

As one of the early MDD adopters in industry, General Motors (GM) has created a domain-specific modeling language, implemented as an internal proprietary metamodel, for Vehicle Control Software (VCS) development. The metamodel defines modeling constructs for VCS development, including physical nodes on which software is deployed and execution frames that are responsible for software scheduling. VCS models conforming to this internal, proprietary metamodel (which we refer to as the GM metamodel) have been used in several vehicle production domains at GM, such as body control which manages the functionality of units like the display and the adaptive cruise control.

Recently, AUTOSAR (the AUTomotive Open System ARchitecture) [14] has been developed as an industry standard to facilitate exchangeability and interoperability of software components from different manufacturers and suppliers. AUTOSAR defines its own metamodel with a well-defined layered architecture and interfaces. Since converging to AUTOSAR is a strategic direction for future modeling activities, transforming GM-specific, legacy models to their equivalent AUTOSAR models becomes essential. Model transformation is considered as a key enabling technology to achieve this convergence objective.

Thus, we develop a model transformation that transforms the GM legacy models into their AUTOSAR equivalents, i.e., the source metamodel is the proprietary GM
metamodel and the target metamodel is the AUTOSAR System Template, version 3.1.5 [15]. To simplify the exercise without losing generality, a subset of the GM metamodel and the AUTOSAR System Template is manipulated in the transformation. Specifically, we focus on the modeling elements related to the software components deployment and interactions, as discussed below.

### 3.2.1 The Source GM Metamodel

Fig. 3.2 illustrates the meta-types in the GM metamodel\(^1\) that represent the physical nodes, deployed software components and their interactions.

![Figure 3.2: The subset of the GM metamodel used in our transformation.](image)

The **PhysicalNode** type specifies a physical unit on which software is deployed. A **PhysicalNode** may contain multiple **Partition** instances, each of which defines a processing unit or a memory partition in a **PhysicalNode** on which software is deployed. Multiple **Module** instances can be deployed on a single **Partition**. The **Module** type defines the atomic deployable, reusable element in a product line and can contain multiple **ExecFrame** instances. The **ExecFrame** type, i.e., an execution frame, models the basic unit for software scheduling. It contains behavior-encapsulating entities, and is responsible for managing services provided or required by the behavior-encapsulating entities. Thus, each **ExecFrame** may provide and/or require **Service** instances, that

\(^1\)The metamodel has been altered for reasons of confidentiality. However, the relevant aspects required for the purpose of this chapter have all been preserved.
model the services provided or required by the ExecFrame.

3.2.2 The Target AUTOSAR Metamodel

The AUTOSAR metamodel is defined as a set of templates, each of which is a collection of classes, attributes, and relations used to specify an AUTOSAR artifact such as software components. Among the defined templates, the System template [15] is used to capture the configuration of a system or an Electronic Component Unit (ECU). An ECU is a physical unit on which software is deployed. When used to represent the configuration of an ECU, the template is referred to as the ECU Extract. Fig. 3.3 shows the metatypes in the ECU Extract that capture software deployment on an ECU. Our transformation manipulates AUTOSAR version 3.1.5.

![AUTOSAR System Template](image)

Figure 3.3: The AUTOSAR System Template containing relevant types used by our transformation.

The ECU extract is modeled using the System type that aggregates SoftwareComposition and SystemMapping elements. The SoftwareComposition type points to the CompositionType type which eliminates any nested software components in a
3.3. **THE GM-TO-AUTOSAR MODEL TRANSFORMATION**

SoftwareComposition instance. The SoftwareComposition type models the architecture of the software components deployed on an ECU, the ports of these software components, and the ports connectors. Each Software component is modeled using a ComponentPrototype, which defines the structure and attributes of a software component; each port is modeled using a PortPrototype, i.e., a PPortPrototype or a RPortPrototype for providing or requiring data and services; each connector is modeled using a ConnectorPrototype. Each ComponentPrototype must have a type that refers to its container CompositionType.

The SystemMapping type binds the software components to ECUs and the data elements to signals and frames (not shown). The SystemMapping type aggregates the SwcToEcuMapping type, which assigns SwcToEcuMapping_components to an EcuInstance. SwcToEcuMapping_components in turn, refer to ComponentPrototype elements. According to AUTOSAR, only one SwcToEcuMapping should be created for each processing unit or memory partition in an ECU.

### 3.3 The GM-to-AUTOSAR Model Transformation

To describe our GM-to-AUTOSAR model transformation, we first demonstrate the transformation rules needed to map between the two metamodels which were defined in consultation with domain experts at General Motors (GM). Then, we discuss the transformation implementation. While code snippets of the transformation are not shown due to confidentiality reasons, we describe the development process and the constructs used to achieve the mapping between the manipulated metamodels.

Our transformation takes three inputs: the source GM metamodel, the target AUTOSAR system template, and an input GM model. The output of the transformation
3.3. THE GM-TO-AUTOSAR MODEL TRANSFORMATION

is an AUTOSAR model.

3.3.1 The Model Transformation Design and Development

Our transformation rules were crafted in consultation with domain experts at General Motors to realize the required mappings between the input and output metamodels. For reasons of confidentiality, we present a simplified version of the actual transformation rules defined.

Let $M$ be the input GM model and $M'$ the to-be-generated output AUTOSAR model. The transformation rules are defined as follows:

1. For every element $\text{physNode}$ of the $\text{PhysicalNode}$ type in $M$, generate an element $\text{sys}$ of the $\text{System}$ type, an element $\text{swcompos}$ of the $\text{SoftwareComposition}$ type, a containment relation $(\text{sys}, \text{swcompos})$, an element $\text{composType}$ of the $\text{CompositionType}$ type, a relation $(\text{swcompos}, \text{composType})$, an element $\text{sysmap}$ of the $\text{SystemMapping}$ type, a containment relation $(\text{sys}, \text{sysmap})$, and an element $\text{ecuInst}$ of the $\text{EcuInstance}$ type in $M'$;

2. For every element $\text{partition}$ of the $\text{Partition}$ type in $M$, generate an element $\text{swc2ecumap}$ of the $\text{SwcToEcuMapping}$ type and a containment relation $(\text{sysmap}, \text{swc2ecumap})$ in $M'$;

3. For every containment relation $(\text{physNode}, \text{partition})$ in $M$, generate a relation $(\text{swc2ecumap}, \text{ecuInst})$ in $M'$;

4. For every element $\text{mod}$ of the $\text{Module}$ type in $M$, generate an element $\text{swc\_comp}$ of the $\text{SwcToEcuMapping\_component}$ type that refers to an element $\text{comp}$ of the $\text{ComponentPrototype}$ type in $M'$;
5. For every containment relation \((\text{partition}, \text{mod})\) in \(M\), generate a containment relation \((\text{composType}, \text{comp})\), a \textit{type} relation \((\text{comp}, \text{composType})\), and a \textit{component} relation \((\text{sw2ecumap}, \text{swc_comp})\) in \(M'\);

6. For every relation \((\text{exframe}, \text{svc})\) of the \textit{provided} type between a \text{exframe} element of the \textit{ExecFrame} type and a \text{svc} element of the \textit{Service} type with a containment relation \((\text{mod}, \text{exframe})\), generate a \textit{pPort} element of the \textit{PPortPrototype} type and a containment relation \((\text{composType}, \text{pPort})\) in \(M'\);

7. For every relation \((\text{exframe}, \text{svc})\) of the \textit{required} type between a \text{exframe} element of the \textit{ExecFrame} type and a \text{svc} element of the \textit{Service} type with a containment relation \((\text{mod}, \text{exframe})\), generate a \textit{rPort} element of the \textit{RPortPrototype} type and a containment relation \((\text{composType}, \text{rPort})\) in \(M'\).

We use the example in Fig. 3.4 to demonstrate the required transformation. Fig. 3.4(a) shows a sample model from the automotive industry that captures the \textit{BodyControl} controller. \textit{Partitions} running on the \textit{BodyControl PhysicalNode} include \textit{SituationManagement} and \textit{HumanMachineInterface}. Each \textit{Partition} may contain multiple \textit{Modules}. The \textit{SituationManagement Partition} contains an \textit{AdaptiveCruiseControl Module} and the \textit{HumanMachineInterface Partition} contains a \textit{Display Module}. Each \textit{Module} runs multiple \textit{ExecFrames} at the same or different rates. The \textit{AdaptiveCruiseControl Module} contains a \textit{ComputeDesiredSpeed ExecFrame} and the \textit{Display Module} contains a \textit{DisplaySetSpeed ExecFrame}. \textit{ExecFrames} invoke \textit{Services} for variable updates. The \textit{ComputeDesiredSpeed ExecFrame} provides a \textit{Service} that is required by the \textit{DisplaySetSpeed ExecFrame}. The expected output AUTOSAR model based on the above mentioned rules is shown in Fig. 3.4(b). The \textit{PhysicalNode} element is mapped to an \textit{EcuInstance} element, a \textit{System} element, a \textit{SystemMapping}
element, a SoftwareComposition element, and a CompositionType element (Rule 1). The Partition elements are mapped to the SwcToEcuMapping elements (Rule 2), each of which has an association with the generated EcuInstance element (Rule 3). The Module elements are mapped to the SwcToEcuMapping_component elements and ComponentPrototype elements that are aggregated by a CompositionType element (Rules 4-5). The ComponentPrototype elements point to their container CompositionType element as their type (Rule 5). Further, the SwcToEcuMapping_component elements are referred to by their corresponding SwcToEcuMapping elements (Rule 5).
3.3. THE GM-TO-AUTOSAR MODEL TRANSFORMATION

The ExecFrame element aggregating a provided Service is mapped to a PPortPrototype element and is aggregated by the CompositionType element (Rule 6). The other ExecFrame element is mapped similarly to a RPortPrototype element (Rule 7).

3.3.2 The Model Transformation Implementation in ATL

We implemented the GM-to-AUTOSAR model transformation using the ATL programming language [48, 75] in the MDWorkbench eclipse based tool [144]. We discussed some details on both MDWorkbech and ATL in a previous study [141]. Since we use terminology from the ATL language in this thesis, we again summarize basic ATL concepts in Appendix B.

Since a non-disclosure agreement prevents us from providing the full details of the GM-to-AUTOSAR transformation, we only summarize the used rules and helpers alongside their functionality in this section. The GM-to-AUTOSAR transformation contains two ATL matched rules (Table 3.1) and 9 functional helpers (Table 3.2) implementing the 7 rules in Section 3.3.1. We also define 6 attribute helpers to access the model attribute values.

<table>
<thead>
<tr>
<th>Matched Rule</th>
<th>Corresponding Rule: Section 3.3.1</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>createComponent</td>
<td>4-5</td>
<td>Maps a Module to a SwcToEcuMapping_component and a ComponentPrototype.</td>
</tr>
<tr>
<td>initSysTemplate</td>
<td>1</td>
<td>Maps a PhysicalNode to a System, a SystemMapping, a SoftwareComposition, and a CompositionType.</td>
</tr>
</tbody>
</table>

Table 3.1: Matched rules, their corresponding rules from Section 3.3.1, and their functionality.

The matched rule createComponent maps a Module element to a SwcToEcuMapping_component element and a ComponentPrototype elements. The matched rule initSysTemp maps a PhysicalNode element to a System element, a SystemMapping element, a SoftwareComposition element, and a CompositionType element by calling
### Table 3.2: Functional helpers, their corresponding rules from Section 3.3.1, and their functionality.

<table>
<thead>
<tr>
<th>Functional Helper</th>
<th>Corresponding Rules: Section 3.3.1</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>initEcuInst</td>
<td>1</td>
<td>Initializes an EcuInstance using the name of a PhysicalNode as an input.</td>
</tr>
<tr>
<td>createSwc2EcuMappings</td>
<td>2-3</td>
<td>Creates Swc2EcuMappings for all Partitions in the input model.</td>
</tr>
<tr>
<td>initSingleSwc2EcuMapping</td>
<td>2-3</td>
<td>Initializes a SwcToEcuMapping using an EcuInstance and a Partition as inputs.</td>
</tr>
<tr>
<td>addComponents</td>
<td>5</td>
<td>Creates the relation between a SwcToEcuMapping and its ComponentPrototypes.</td>
</tr>
<tr>
<td>getAllPPortsInEcu</td>
<td>6</td>
<td>Creates a PPortPrototype for any ExecFrame that has at least one provided Service.</td>
</tr>
<tr>
<td>createPPort</td>
<td>6</td>
<td>Initializes one PPortPrototype.</td>
</tr>
<tr>
<td>getAllRPortsInEcu</td>
<td>7</td>
<td>Creates a RPortPrototype for any ExecFrame that has at least one required Service.</td>
</tr>
<tr>
<td>createRPort</td>
<td>7</td>
<td>Initializes one RPortPrototype.</td>
</tr>
<tr>
<td>getAllSWCinEcu</td>
<td>5</td>
<td>Creates the containment relation between CompositionTypes and ComponentPrototypes.</td>
</tr>
</tbody>
</table>

The 9 functional helpers to implement rules 1-3 and 5-7. The helper *initEcuInst* initializes an EcuInstance element. The helper *initSingleSwc2EcuMapping* initializes a SwcToEcuMapping instance. The helper *createSwc2EcuMappings* creates a list of Swc2EcuMapping elements corresponding to all the Partition elements in the input model. The helper *getAllSwcInEcu* creates the containment relation between the CompositionType elements and the ComponentPrototype elements. The helper *addComponents* creates the relation between the SwcToEcuMapping elements and their corresponding ComponentPrototype elements. The helper *getAllPPortsInEcu* creates a PPortPrototype element using the helper *createPPort* for ExecFrames with at least one provided Service. Similar helpers generate RPortPrototype elements.

The relationships between the outputs of the two matched rules are built using the ATL predefined function *resolveTemp* which connects the ComponentPrototype elements created by the createComponent matched rule to the CompositionType element created by the initSysTemp matched rule. The resolveTemp function allows a rule to reference the elements that are yet to be generated by other rules at runtime.
Implementation of the seven mapping rules in Section 3.3.1 as an ATL transformation composed of two matched rules, 9 functional helpers, and 6 attribute helpers (as described in this section) was not a straightforward process; implementation of the transformation was iterative and incremental. First, a simple GM model was created using the MDWorkbench model editor. Then, one (or more) transformation rule/s or helper/s were implemented to transform the input GM model into an equivalent output AUTOSAR model. The output AUTOSAR model was then manually checked to ensure that the transformation performed the required mapping. If the output model was found to be correct, the process was repeated with additional metatypes in the input model and additional rules or helpers in the transformation to process these metatypes. If the output model contained errors, the transformation was analyzed, and any erroneous rules or helpers were fixed. Throughout the development process, rules were deconstructed and reconstructed on-the-go as we were trying to improve the modularity and readability of the transformation.

3.4 Summary

In this chapter, we presented a model transformation that migrates GM, legacy VCS design models to their equivalent AUTOSAR models. We discussed the transformation context in the development process, the manipulated metamodels, the design and development of the transformation, and the implementation details. We previously discussed technical issues related to the development of the transformation in more detail [141], e.g., the process of choosing a suitable tool and language for implementing the transformation, the challenges encountered in developing the transformation,
and limitations in existing transformation development environments that need to be addressed by tool developers.

This GM-to-AUTOSAR transformation will be used in Chapter 4 to experiment with state-of-the-art transformation verification tools and get a better understanding of the limitations of these tools. This transformation will also be used in Chapter 6 to evaluate the verification tool that we redevelop, extend, and describe in Chapter 5.
Chapter 4

Experimenting with State-of-the-Art Model Transformation Verification Tools

We verify our GM-to-AUTOSAR model transformation (described in Chapter 3) using two state-of-the-art model transformation verification tools (i.e., one dynamic verification tool and one static verification tool). Experimenting with such tools will give us a better understanding of the limitations of existing verification approaches that yet need to be addressed by researchers. We aim to address these limitations in a verification tool that we redevelop and describe later in Chapter 5.

From the dynamic model transformation verification tools, we use an existing black-box testing tool, the MetaModel Coverage Checker (MMCC) [71, 53], to verify the GM-to-AUTOSAR transformation in Section 4.1. From the static model transformation verification tools, we use an automated formal verification tool [34, 35] (i.e., based on Type II formal methods, explained in Section 2.2.2) to verify the GM-to-AUTOSAR transformation in Section 4.2. In both Sections 4.1 and 4.2, we report on the obtained verification results and the strengths and limitations of the two tools. We summarize the chapter and the findings in Section 4.3.
4.1 Black-Box Testing for Model Transformation Verification

We use model transformation testing \cite{138} to verify the GM-to-AUTOSAR transformation (Chapter 3). As discussed in Section 2.3.5, testing executes a model transformation on input test models or a test suite and validates that the generated, actual output model or code matches the expected output model or code \cite{64}. The test suite is built by defining test adequacy criteria and building a test suite that achieves coverage of the adequacy criteria \cite{138}. In general, defining test adequacy criteria, and hence testing, can follow a black-box or a white-box approach. Black-box testing assumes that the implementation of the transformation of interest is a black-box and builds a test suite based on the specification of the transformation (i.e., source metamodel or contracts). On the other hand, white-box testing assumes that the implementation of the transformation of interest is available and builds a test suite based on the implementation of the transformation.

For our work, we use black-box testing to validate our GM-to-AUTOSAR transformation. As described in Section 2.3.5, testing is composed of three phases: test case generation, (optional) evaluation of the test suite, and building the oracle function. We use the Metamodel Coverage Checker (MMCC) tool \cite{71} to facilitate the first step of test case generation. MMCC guides the user in building a test suite based on a predefined test adequacy criterion and a source metamodel. MMCC was implemented in Kermeta \cite{70} as part of a study by Fleurey et al \cite{53}. We do not carry out the second optional testing step, i.e., evaluation of the test suite. The third step, i.e., building the oracle function, was done by manual inspection of the transformation’s outputs when run on the test suite. This step was conducted manually due to the unavailability of expected output models with which we can compare the
actual output models.

Section 4.1.1 describes the inner workings of MMCC. Section 4.1.2 discusses the application of MMCC to the GM-to-AUTOSAR transformation and the testing results. Finally, Section 4.1.3 summarizes the strengths and limitations of using black-box testing for transformation verification, based on our experience with MMCC.

### 4.1.1 MetaModel Coverage Checker (MMCC)

MMCC [71, 53] runs on two phases. In the first phase, the user specifies the source metamodel and an adequacy criterion as inputs. In this phase, MMCC uses category-partitioning [107] to partition the values of multiplicities and attributes (in the source metamodel) of type integer, string, or boolean into ranges as follows:

- Integer attribute values and multiplicity values are partitioned into three ranges: \( \{0\}, \{1\}, \text{ and } \{>1\} \).
- String attribute values are partitioned into two ranges: \( \{"\} \) and \( \{"+"\} \) (i.e. an empty string and a non-empty string).
- Boolean attribute values are partitioned into two ranges: \{true\} and \{false\}.

We update MMCC to generate partitions for attributes that are of types other than integer, string, or boolean. For example, float attributes were partitioned into three ranges: \( \{0\}, \{(0,1]\}, \text{ and } \{>1\} \).

Using the generated partitions and the specified adequacy criterion, MMCC generates object fragments and model fragments. An object fragment is a template for a class object that specifies constraints on the values of the attributes and multiplicities of objects from the corresponding class. A model fragment is a template for an input
4.1. BLACK-BOX TESTING FOR MODEL TRANSFORMATION VERIFICATION

A test model that contains one or more object fragments. A model fragment is satisfied by a test model if the objects in the test model satisfy the object fragments in the model fragment.

In the second phase, the user specifies the location of a test suite and MMCC evaluates the test suite by identifying how many model fragments generated in the first phase were satisfied by the test suite. MMCC further generates a summary of the missing model fragments in the test suite to guide the user in building additional test models that satisfy these missing model fragments.

4.1.2 Results

For the first phase, we specified two inputs to run MMCC: the source GM metamodel (Section 3.2.1) of our GM-to-AUTOSAR transformation and the AllPartitions criterion. The AllPartitions criterion is a criterion implemented in MMCC and mandates that values from all ranges of each property or multiplicity partition should be represented simultaneously in the same model fragment. For example, for an integer attribute, one model fragment mandates that the attribute should have values from the three integer ranges ($\{0\}$, $\{1\}$, and $\{>1\}$) in a single input model. In this phase, MMCC generated 196 partitions for 196 different attributes and multiplicities values. Accordingly, 196 model fragments were generated for the AllPartitions criterion.

Besides the AllPartitions criterion, the AllRanges criterion was also implemented in MMCC. The AllRanges criterion mandates that values from each range of each property or multiplicity partition should be represented in a model fragment. We used the AllPartitions criterion instead of the AllRanges criterion since it subsumes the AllRanges criterion, i.e., a test suite that satisfies the AllPartitions criterion also
satisfies the $AllRanges$ criterion, but the inverse is not true.

We did not run the second phase of MMCC since we started off with an empty test suite. Thus we need to build a test suite with models that satisfy the 196 model fragments. Having 196 model fragments implies that the test suite can contain at most 196 models to satisfy the $AllPartitions$ criterion. However, one model can cover more than one model fragment at a time. Thus, we manually built a test suite of 100 test models to cover the 196 model fragments.

Our model transformation was executed using the generated test suite. For each test model in the test suite, the corresponding output model was verified by manually checking whether the output AUTOSAR model is a valid equivalent of the input GM model. The transformation was found to produce the expected output models for the 100 input test models.

Actual GM models were not used for testing since many of the actual GM models did not conform to the GM metamodel. They were built using IBM Rational Rhapsody [67] which allows building models without mandating that these models be valid instances of a specific metamodel (i.e., Rhapsody does not check conformance of the GM models to the GM metamodel). Thus, migration from GM models to their equivalent AUTOSAR models can be done in two ways. The first alternative is to manually build the AUTOSAR equivalents of all the models to be migrated. The major drawback of this alternative is that different engineers may have different understandings of AUTOSAR and the migration may be inconsistent for different models. The second alternative is to update the GM models to ensure that they conform to the GM metamodel and then using our transformation to migrate all GM models (conforming to the GM metamodel) to their AUTOSAR equivalents in an
automated, consistent way. The second alternative is easier to adopt since changing
the GM models to conform to the GM metamodel can, in many cases, involve minor
changes (e.g. updating an association, adding an attribute name) which is much sim-
pler than building AUTOSAR models from scratch (as in the first alternative) and
ensuring that they convey the intended meaning.

4.1.3 Strengths and Limitations of Black-Box Testing

We identify three strengths of using black-box testing for model transformation veri-
fication. First, unlike formal verification approaches that use formalizations such as
Maude [39] (Appendix A.5), testing does not require that the user has a thorough
knowledge of the formalization. Second, testing has the advantage of uncovering bugs
while maintaining a low computational complexity [60]. Third, several studies that
used formal transformation verification (as opposed to testing) proposed reimplement-
ing transformations in other formalizations such as Petri Nets [111] and Maude [39]
(Appendix A) to use their verification techniques [78, 155, 27, 127]. Reimplement-
ing industrial-size transformations in a different formalization can be infeasible due
to time and money constraints. MMCC is one of the few publicly available tools
that can validate transformations in any formalization (including ATL) since it is a
black-box testing tool (i.e., transformation-language independent).

We identify two limitations of using black-box testing for model transforma-
tion verification. First, after manually examining the model fragments generated
by MMCC and the corresponding test models built to satisfy the model fragments,
we found that only 45 model fragments out of the 196 actually trigger any rule in
our GM-to-AUTOSAR model transformation. The generation of redundant model
fragments and the possibility of the test suite not triggering all the rules in the transformation are due to the nature of black-box testing in general; test cases are generated independent of the model transformation implementation. Second, MMCC helped provide a systematic way to generate a test suite, but the actual generation of the test suite, execution of the transformation using the test models, and validation of the generated output models was performed manually, and hence was time consuming and error prone (i.e., manual validation of the output models did not uncover two bugs in the transformation that were uncovered by the tool we describe in Section 4.2). For testing to scale up to industrial size transformations and models, it is necessary to increase the level of automation in generating the test suite, executing the transformation of interest using the test suite, and evaluating the testing results (e.g., using model differencing as described in Section 2.3.5).

4.2 Type II Formal Methods for Model Transformation Verification

We use a light-weight, automated verification approach to verify the GM-to-AUTOSAR transformation (Chapter 3). The basic approach was presented by Büttner et al. [34, 35] and allows reasoning about the correctness of ATL [75] transformations. The approach is based on checking the satisfiability of a relational transformation representation, or a transformation model, with respect to well-formedness OCL constraints. An implementation of this approach is available as a prototype for the ATL language.

Section 4.2.1 introduces the applied verification approach and prototype. Section 4.2.2 describes the constructs used to reimplement the GM-to-AUTOSAR transformation. Section 4.2.3 describes the different kinds of constraints formulated for verification using the verification prototype. Section 4.2.4 summarizes the verification
4.2. TYPE II FORMAL METHODS FOR MODEL
TRANSFORMATION VERIFICATION

results and investigates the performance of the prototype. Section 4.2.5 discusses the
strengths and limitations of the used approach.

4.2.1 Verification Methodology

We use the automated verification approach presented by Büttner et al. [34, 35] to
verify the GM-to-AUTOSAR transformation. In short, the approach translates an
ATL transformation $T$, its source metamodel $MM_{src}$, and its target metamodel $MM_{tar}$
into a combined model, or a transformation model, consisting of $MM_{src}$, $MM_{tar}$, and
additional model elements that represent the transformation rules. Moreover, a set
$Sem$ of OCL constraints is generated for the transformation model that characterizes
the execution semantics of the ATL rules. For declarative and non-recursive ATL
rules, the constraints describe the ATL semantics one-to-one, i.e., each valid instance
of the transformation model corresponds to an execution of the transformation and
vice versa.

This transformation model is used to check the partial correctness of the transfor-
mation with respect to properties specified as OCL constraints over the source and/or
the target metamodels, by checking if there exists a counterexample within a specific
scope (i.e., maximum number of objects per class). More specifically, for a set of
transformation preconditions (or assumptions) $Pre_{1}, \ldots, Pre_{n}$ and a set of postcon-
ditions (or assertions) $Post_{1}, \ldots, Post_{m}$, we want to show that for each instance $M$
of the transformation model,

\[
(Sem_1 \text{ and } Sem_2 \text{ and } \ldots \text{ and } Sem_k \text{ and } Pre_1 \text{ and } Pre_2 \text{ and } \ldots \text{ and } Pre_n) \text{ implies } (Post_1 \text{ and } Post_2 \text{ and } \ldots \text{ and } Post_m) \tag{4.1}
\]

holds. Eqn. 4.1 can be expressed equivalently as follows: For each postcondition \(Post_i\) (1 ≤ i ≤ m), the following formula must be unsatisfiable (i.e., there is no model \(M\) under which the formula is true):

\[
Sem_1 \text{ and } \ldots \text{ and } Sem_k \text{ and } Pre_1 \text{ and } \ldots \text{ and } Pre_n \text{ and not}(Post_i) \tag{4.2}
\]

Fig. 4.1 illustrates this using a simple example. In the upper part, we have an ATL transformation (c) over the shown source and target metamodels (a) and (b). The transformation maps the A-B structure to the C-D structure, but creates an additional D object when mapping an ‘empty’ A object. The middle part shows the transformation model of this transformation. In the class diagram (d), each of the three rules is translated into a trace class and connected to the source and target classes according to the from and to patterns of the rule. The OCL constraints (e) capture the execution semantics of the transformation such as the matching of rule R1, the binding of primitive and object-typed properties, and the controlled creation of target objects. Some pre-/post- conditions are shown in (f) and (g), respectively.

To verify that, for example, postcondition \(Post_i\) is implied by the transformation (given the preconditions), we have to check that Eq. (4.2) is unsatisfiable. This can be checked using metamodel satisfiability checkers, or model finders, such as the
4.2. TYPE II FORMAL METHODS FOR MODEL TRANSFORMATION VERIFICATION

USE Validator [79] which is publicly available [154]. The USE Validator translates the UML transformation model and its OCL constraints into a relational logic formula and employs the SAT-based solver Kodkod [151] to check the unsatisfiability of Eq. (4.2) for each of the post-conditions $Post_i$ within a given scope. Thus, the verification approach has four different representations of the problem space, (i) ATL and OCL, (ii) OCL, (iii) relational logic, and (iv) propositional logic (for the SAT solver).

The whole chain was implemented as a verification prototype [34], as shown in Fig. 4.2. The prototype converts the manipulated metamodels and the ATL transformation into a transformation model using the ATL-to-TM transformation [34]. The ATL-to-TM transformation was implemented as a higher-order ATL transformation [150], i.e., a transformation from Ecore metamodels and an ATL transformation

![Diagram](image)

(a) Source MM  (b) Target MM  (c) ATL transformation  (d) Transformation model  (e) OCL constraints for ATL semantics (excerpt)

![Context](image)

(f) Preconditions  (g) Postconditions

Figure 4.1: Transformation model example
to an Ecore transformation model (where the Ecore model can be annotated with OCL constraints). The ATL-to-TM transformation automatically generates the $Sem$ constraints from the ATL transformation as well as the pre- and postconditions from the structural constraints in the source and target metamodels. Since the USE validator has a proprietary metamodel syntax, a converter from the Ecore transformation model to a USE specification and a default search space configuration was created. Besides the automatically generated constraints in the USE specification, additional constraints to be verified can be added manually. The search space configuration is a file specifying the scopes and ranges for the attribute values. In the search space configuration, individual invariants or constraints can be disabled or negated.

![Figure 4.2: The tool chain used to perform the transformation verification.](image)

**Steps to verify a postcondition using the prototype:** To check Eqn. (4.2) for a postcondition, we negate the respective postcondition and disable all other postconditions in the generated search space configuration (Fig. 4.2), and then run USE. If USE reports ‘unsat’, this implies that there is no input model in the search space for which the transformation can produce an output model that violates the postcondition. If there exists a counterexample, USE provides the object diagram of the counterexample which can be analyzed using many browsing features of the tool.
4.2. TYPE II FORMAL METHODS FOR MODEL TRANSFORMATION VERIFICATION

4.2.2 Reimplementation of the GM-to-AUTOSAR Model Transformation

We use the prototype [34, 35] described in Section 4.2.1 to verify our GM-to-AUTOSAR transformation (described in Chapter 3). Since the verification prototype can only verify ATL transformations composed of declarative matched rules and non-recursive lazy rules, we have changed the implementation described in Section 3.3.2 to be completely declarative and compatible with the format required by the prototype. The final reimplementation is intended to achieve the same mapping described in Section 3.3.1 as the original implementation.

The first implementation of the GM-to-AUTOSAR transformation (described in Section 3.3.2) used two ATL matched rules, 9 functional helpers, and 6 attribute helpers to implement the required mapping between the two metamodels. After reimplementing the transformation to be completely declarative, the new GM-to-AUTOSAR transformation was composed of three matched rules and two lazy rules. Although we had to reimplement the transformation to use the verification prototype, we point out that the new declarative implementation is simpler and more readable. The rules implemented are listed in Table 4.1 together with the types of the rules, the input element matched by the rule, and the output elements generated by the rule.

As described in Section 3.3.2, the relationships between the outputs of the matched rules are built using the ATL predefined function \texttt{resolveTemp}. The \texttt{resolveTemp} function allows a rule to reference the elements that are yet to be generated by another rule at runtime. For example, the \texttt{resolveTemp} function was used to connect the \textit{SwcToEcuMappings} created by the \texttt{initSingleSwc2EcuMapping} matched rule to the \textit{SystemMappings} created by the \texttt{initSysTemp} matched rule. Further, the matched
4.2. TYPE II FORMAL METHODS FOR MODEL TRANSFORMATION VERIFICATION

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Rule Name</th>
<th>Input Types</th>
<th>Output Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matched Rule</td>
<td>createComponent</td>
<td>Module</td>
<td>SwCompToEcuMapping, ComponentPrototype</td>
</tr>
<tr>
<td>Matched Rule</td>
<td>initSysTemp</td>
<td>PhysicalNode</td>
<td>System, SystemMapping, SoftwareComposition, CompositionType, EcuInstance</td>
</tr>
<tr>
<td>Matched Rule</td>
<td>initSingleSwc2EcuMapping</td>
<td>Partition</td>
<td>SwcToEcuMapping</td>
</tr>
<tr>
<td>Lazy Rule</td>
<td>createPPort</td>
<td>ExecFrame</td>
<td>PPortPrototype</td>
</tr>
<tr>
<td>Lazy Rule</td>
<td>createRPort</td>
<td>ExecFrame</td>
<td>RPortPrototype</td>
</tr>
</tbody>
</table>

Table 4.1: The types of ATL constructs used to reimplement the transformation, their designated names, and their input and output element types.

The rule `initSysTemp` calls the two lazy rules and assigns the union of the lazy rules’ outputs to the ports of the `CompositionType` produced by the `initSysTemp` rule.

4.2.3 Formulation of OCL Pre- and Postconditions

The verification prototype can be used to define OCL postconditions either on elements of the target metamodel only (which we refer to as **target invariants**), or they can relate the elements of the source and target metamodels (which we refer to as **transformation contracts**). Usually, a transformation contract specifies an implication of the form ‘when an input has a property then its corresponding output has a property’. The OCL preconditions are propositions about the input that are assumed to always hold.

The preconditions of the GM-to-AUTOSAR transformation are the multiplicity and composition constraints automatically extracted by the prototype from the GM metamodel as OCL constraints. The OCL postconditions that we formulated are summarized in Table 4.2. We divide the formulated postconditions into four categories: **Multiplicity Invariants**, **Uniqueness Contracts**, **Security Invariants**, and **Pattern Contracts**. For each constraint in Table 4.2, we add to the beginning of its formulation an abbreviation (e.g., \( (M1), (U2) \)) that we will use to refer to the constraint. The
4.2. TYPE II FORMAL METHODS FOR MODEL TRANSFORMATION VERIFICATION

Multiplicity Invariants were automatically generated by the prototype. All the other postconditions were manually formulated.

Multiplicity Invariants ensure that the transformation does not produce an output that violates the multiplicities in the AUTOSAR metamodel (Fig. 3.3). As described in Section 4.2.1, the prototype generates a USE specification with a multiplicity invariant for each multiplicity in the AUTOSAR metamodel. Ideally, we would check the satisfiability of all the multiplicity invariants generated for the AUTOSAR metamodel. Since our transformation manipulates a subset of the metamodels, we only check multiplicity invariants for output elements affected by our transformation. We identify six of the generated multiplicity invariants that are affected by our transformation. 

\( (M_1) \) ensures that each \textit{CompositionType} is associated to more than one \textit{ComponentPrototype} through the \textit{component} association. \( (M_2) \) ensures that each \textit{SoftwareComposition} is associated with one \textit{CompositionType} through the \textit{softwareComposition} association. The rest of the multiplicity invariants can be interpreted similarly.

Uniqueness Contracts require the output element (of type Y) generated by a rule to be uniquely named (by the \textit{shortName} attribute) within its respective scope if the corresponding input element (of type X) matched by the rule is uniquely named (by the \textit{Name} attribute) within its scope, too. For example, in Section 4.2.2, we discussed that the matched rule \texttt{createComponent} maps \texttt{Module}s to \texttt{ComponentPrototype}s. Thus, \( U_1 \) mandates that the \texttt{ComponentPrototype}s generated by the transformation are uniquely named, if the corresponding \texttt{Module}s are uniquely named, too. The rest of the Uniqueness Contracts are similar and ensure the uniqueness of the output elements of each rule described in Section 4.2.2 if their corresponding input elements
are unique, too.

The only security invariant defined, $S1$, mandates that within any System element, all its composite SwcToEcuMappings must refer to ComponentPrototypes that are contained within the CompositionType lying under the same System element (refer to Fig. 3.2). Thus, this invariant assures that any ECU configuration (modeled by a System element) is self contained and does not refer to any ComponentPrototype that is not allocated in that ECU configuration.

Pattern contracts require that if a certain pattern of elements is in the input model, then a corresponding pattern of elements must be in the output model. Pattern contracts also mandate that corresponding elements in the input and output patterns must have the same name. $P1$ mandates that if a PhysicalNode is connected to a Service through the provided association (in the input model), then the corresponding System element will eventually be connected to a PPortPrototype. $P1$ also ensures that the names of the PhysicalNode and the System are equivalent and that the names of the ExecFrame (containing the Service) and the PPortPrototype are equivalent. The contract $P2$ is similar to $P1$ but manipulates required Services and RPortPrototypes instead.

Since invariants are constraints on target metamodel elements, the Multiplicity and Security invariants are specified within the context of their respective AUTOSAR elements. Since contracts are constraints on the relationships between the source and target metamodel elements, they do not relate to an AUTOSAR element per se. Thus, we add a class to the USE specification file, Global, which is used as the context of the Uniqueness and Pattern contracts.
4.2.4 Results

We discuss the results of verifying the OCL constraints defined in Section 4.2.3 using the verification prototype [34, 35]. We show how the prototype was able to uncover bugs in the GM-to-AUTOSAR transformation and we discuss the performance of the prototype.

Verifying the Formulated OCL Constraints

Using the verification prototype, we generated a USE specification and a search space configuration, as shown in Fig. 4.2. After adding the constraints (Table 4.2) to the USE specification, we ran the USE tool once for each constraint.

Out of the 18 constraints defined in Table 4.2, two multiplicity invariants were found to be violated by the transformation: CompositionType_component and SwcToEcuMapping_component. In other words, our transformation can generate a CompositionType with no ComponentPrototypes and/or a SwcToEcuMapping with no ComponentPrototypes. Both of these possible outputs violate the multiplicities defined in the AUTOSAR metamodel (Fig. 3.3). The counterexamples were found by USE within a scope of one object per concrete class.

We show the counterexample generated for the CompositionType_component invariant in Fig. 4.3. The counterexample shows that the rule initSysTemp maps a PhysicalNode to five elements, one of which is CompositionType. Since the rule does not have any restrictions on the generated CompositionType, it was created without associating it to any ComponentPrototype through the component association. The counterexample for the SwcToEcuMapping_component invariant was similar showing that the initSingleSwc2EcuMapping rule creates a SwcToEcuMapping element
4.2. TYPE II FORMAL METHODS FOR MODEL TRANSFORMATION VERIFICATION

Figure 4.3: The counterexample generated for the `CompositionType.component` multiplicity invariant.

without mandating that it is associated to any `SwCompToEcuMapping.component` element through the `component` association.

After examining the two counterexamples generated by USE for the two violated multiplicity invariants, we identified two bugs in two of the rules shown in Table 4.3: `initSysTemp` and `initSingleSwc2EcuMapping`. The bold, underlined text are the updates to the rules that fix the two bugs. `initSysTemp` initially mapped a `PhysicalNode` to many elements, including a `CompositionType` that must contain at least one `ComponentPrototype`. If the `PhysicalNode` did not have any `Module` in any of its `Partitions`, then the created `CompositionType` will not contain any `ComponentPrototypes`. Thus we added a matching constraint to the `PhysicalNode` matched by the rule to ensure that any of its `Partitions` must contain at least one `Module`. Similarly, `initSingleSwc2EcuMapping` initially mapped a `Partition` to a `SwcToEcuMapping` that must contain at least one `SwCompToEcuMapping.component`. If the `Partition` did not have any `Module`, then the created `SwcToEcuMapping` will not contain any `SwCompToEcuMapping.component`. Thus, we added a matching constraint to the `Partition` matched by the rule to ensure that it must contain at least one `Module`. 
4.2. TYPE II FORMAL METHODS FOR MODEL TRANSFORMATION VERIFICATION

Table 4.3: The two rules that required updates to address the two violations of multiplicity invariants.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>initSysTemp</strong></td>
<td>from ph: GM!PhysicalNode (ph.partition→exists(p</td>
</tr>
<tr>
<td><strong>initSingleSwc2EcuMapping</strong></td>
<td>from p:GM!Partition((GM!PhysicalNode.allInstances()→one(ph</td>
</tr>
</tbody>
</table>

The 18 constraints were reverified on the updated transformation, and were all found to be satisfied.

Performance of the Verification Approach

To explore the performance of the verification prototype, we used the prototype to verify the 18 constraints (Table 4.2) for different scopes (i.e., the maximum number of objects per concrete class in the search space). We show the results for scopes 6, 8, 10, and 12. In our tests, we used the same scope for all classes, although they could be set individually. Since our transformation model has 1586 classes, a scope of \( n \) generates a model with \( 1586n \) potential elements (and their corresponding associations and attribute values). All experiments where run on a standard laptop.
4.2. TYPE II FORMAL METHODS FOR MODEL TRANSFORMATION VERIFICATION

at 2.50 GHz and 16 GB of memory, using Java 7, Kodkod 2.0, and Glucose 2.1.

For each combination of constraint and scope, the prototype generates two time values: the time the prototype takes to translate the relational logic formula into a propositional formula (i.e., translation time) and the time the SAT solver takes to solve the formula (i.e., constraint solving time).

We show these two time values (in seconds) in Table 4.4. Each column represents the time intervals for each of the 18 constraints, where the Constraint Abbreviation is the abbreviation given to each constraint in Table 4.2 (e.g., (M1) and (U5)). Each row represents the time intervals for a different scope. Thus, each cell within the table shows the translation time and the constraint solving time of a certain constraint at a specific scope.

<table>
<thead>
<tr>
<th>Scope</th>
<th>U1</th>
<th>U2</th>
<th>U3</th>
<th>U4</th>
<th>U5</th>
<th>U6</th>
<th>U7</th>
<th>U8</th>
<th>U9</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>76\1/25</td>
<td>76\1/19</td>
<td>76\1/22</td>
<td>76\1/7</td>
<td>77\1/19</td>
<td>76\1/24</td>
<td>76\1/7</td>
<td>76\1/7</td>
<td>74\1/5</td>
</tr>
<tr>
<td>8</td>
<td>169\1/74</td>
<td>165\1/79</td>
<td>168\1/106</td>
<td>165\1/37</td>
<td>168\1/85</td>
<td>171\1/68</td>
<td>167\1/38</td>
<td>166\1/57</td>
<td>169\1/45</td>
</tr>
<tr>
<td>10</td>
<td>279\1/165</td>
<td>280\1/188</td>
<td>279\1/210</td>
<td>281\1/114</td>
<td>277\1/211</td>
<td>280\1/207</td>
<td>281\1/147</td>
<td>282\1/170</td>
<td>279\1/206</td>
</tr>
<tr>
<td>12</td>
<td>455\1/976</td>
<td>434\1/643</td>
<td>431\1/623</td>
<td>428\1/322</td>
<td>426\1/827</td>
<td>428\1/616</td>
<td>425\1/584</td>
<td>427\1/604</td>
<td>430\1/501</td>
</tr>
</tbody>
</table>

Table 4.4: Translation Constraint Solving times (seconds) for the 18 constraints on different scopes. For a scope of 12, the verification of S1 did not terminate in a week.

Two observations can be made from Table 4.4. First, despite the exponential complexity of checking boolean satisfiability, we could verify the postconditions for scopes up to 12 in most of the cases. Besides the verification of S1 that did not finish for scope 12, the longest constraint solving time was for S1 in scope 10 (just over an hour). Although we have no proof that no bugs will appear for bigger scopes, we are
confident that a scope of 12 was sufficient to uncover any bugs in our transformation with respect to the defined constraints. In fact, the two bugs that were uncovered and fixed were found at a scope of one.

Second, the translation times are larger than expected and grow mostly polynomially. This can be attributed to the approach used by Kodkod to unfold a first-order relational formula into a set of clauses in conjunctive normal form (CNF), given an upper bound for the relation extents [151]. While transforming a formula into CNF grows exponentially with the length of the formula, it only grows polynomially with the scope in our case (as the formula’s length does not change significantly). For example, each pair of nested quantifiers will generate a number of clauses that grows quadratically with the scope. The relational logic constraints generated implicitly by USE for all associations expand similarly. This justifies why the two pattern contracts (i.e., P1 and P2) show the highest translation times; they have the most quantifiers of the 18 constraints.

Using an incremental SAT solver would improve the performance of the prototype. Since most of the generated Boolean formula is the same for all the 18 constraints (i.e., the encoding of classes, associations, multiplicities, and preconditions), we expect that the translation (i.e., the first number in each cell of Table 4.4) can be done once for the entire verification process; except for P1 and P2 which differ in their high number of nested quantifiers.

4.2.5 Strengths and Limitations of the Verification Approach

Strengths of the Verification Approach We identify two strengths of the verification approach [34, 35]. First, the used approach provides a fully automated
translation from ATL transformations and their constrained metamodels to OCL and relational logic. The approach further provides a fully automated verification of the generated translation. Even when applied to a realistic case study, the approach scaled to a scope that was large enough to strongly suggest that the analysis did not overlook a bug in the transformation due to the boundedness of the underlying satisfiability solving approach. If we wanted to perform the same verification on a Java implementation of the transformation, we would require equally rich class and operation contracts for, say, Ecore in JML [74]. To the best of our knowledge, no research has explored automatically inferring such contracts. Even then, we expect that the user would have to explicitly specify loop invariants if the transformation contains non-trivial loops, like the loops in our transformation.

Second, the verification prototype translates a substantial subset of ATL for verification, i.e., all rules except for imperative blocks, recursive lazy rules, and recursive query operations other than relational closures. Thus, the approach takes advantage of the ways declarative, rule-based transformation languages (e.g., ATL) provide to iterate over the input model without requiring recursion or looping. This simplifies verification by, for instance, obviating the need for loop invariants. Although this subset of ATL is not Turing-complete, it can be used to implement many non-trivial transformations. We have statically checked the 131 transformations (comprising 2825 individual rules) in the ATL transformation zoo [56], and 83 of them fall into the described fragment of ATL, i.e., the transformations neither use recursive rules nor imperative features. Of the remaining 48 transformations, 24 of them that use imperative blocks but no recursion could be expressed declaratively, too. Thus, the
verification prototype benefited from the conceptual simplicity of the declarative fragment of ATL compared to a general-purpose programming language such as Java.

Limitations of the Verification Approach  We identify two limitations of the verification approach [34, 35].

Correctness of ATL-to-relational-logic translation: Extensive testing and inspection was used by the authors of the approach to ensure that all steps involved in the translation of ATL and OCL to first-order relational logic are correct. However, in the absence of a formal semantics of ATL and OCL, a formal correctness proof is impossible and the possibility of a bug in the translation remains. This should be taken into account before using the prototype in the context of safety-critical systems.

Bounded search approach: All verification approaches based on a bounded search space cannot guarantee correctness of a transformation because the scopes experimented with may have been too small. The maximum scope sufficient to show bugs in a transformation is transformation-dependent. For example, a transformation with a multiplicity invariant that requires a multiplicity to be 10, will require a scope of 11 to generate a counterexample for that invariant, if any. With respect to our case study, we are confident that a scope of 5 is sufficient to detect violations of the given constraints; we ran analyses with scopes up to 12, because we wanted to study the performance of the approach. Real proofs of unsatisfiability can be created using SMT solvers and quantifier reasoning [32], but the problem is generally undecidable (i.e., the SAT solver does not terminate on all transformations), and the mapping presented in [32] does not yet cover all language features used in the verification prototype that we used (Section 4.2.1). Further, the authors of the approach have not yet applied any a priori optimizations of the search problem, e.g., metamodel
pruning [143], which they plan to apply for future work.

4.3 Summary

In this chapter, we verified our GM-to-AUTOSAR transformation (described in Chapter 3) using both a dynamic transformation verification tool (Section 4.1) and a static transformation verification tool (Section 4.2). The experiments are intended to give us a better idea of the limitations of existing verification approaches, and hence, help us in building a tool that addresses these limitations.

For dynamic transformation verification, we used a black-box testing tool (i.e., MMCC [71, 53]) to verify our transformation in Section 4.1. We discussed the test case generation criterion used and we reported on the results. Based on manual evaluation of the outputs, the transformation was found to be correct. To the best of our knowledge, no other study discussed testing industrial transformations. Fleurey et al. [54] mentioned that testing was used to verify their migration transformation but the study did not discuss details of the testing process (i.e., the used test case generation criteria, the number of generated test cases, and the results). Work on transformation testing can be extended by automating steps in the testing process, e.g., the test suite generation, test suite assessment using mutation analysis [101], execution of the transformation on the test suite, and evaluation of the transformation's outputs. White-box testing (e.g., [81]) can also be used for verification.

For static transformation verification, we used an automated, formal transformation verification prototype [34, 35] to verify our transformation in Section 4.2. We discussed the verification methodology used, the reimplementation of the transformation to be compatible with the prototype’s expected input format, the formulated
properties of interest, and the verification results. The prototype was able to uncover two bugs in the transformation that violated two multiplicities in the AUTOSAR metamodel. Further, the Translation times and the Constraint Solving times were found to grow exponentially with the scope. Nonetheless, analysis of the transformation in sufficiently large scopes (up to 12) was possible. We point out that while the GM-to-AUTOSAR transformation is not exceptionally large (in the number of rules), the corresponding metamodels are. Together, they comprise 1586 classes, 897 associations, and 371 multiplicity constraints. Since even types not used by the transformation are relevant for the prototype (due to constraints that relate them), the prototype deals with large potential instances. For future work, verification using the prototype can be extended in three ways. First, other transformations should be verified using the prototype to have a better idea of its scalability. We used a transformation that manipulates metamodels that are considered large on an industrial scale. The transformation, although far from being trivial, does not fully manipulate the two metamodels. Thus, we still need to investigate the performance on larger transformations. Second, the prototype can be adapted to use incremental SAT solvers in the bounded search to improve the prototype’s performance, as suggested in Section 4.2.4. Third, the prototype can be extended to prune the manipulated metamodels or the transformation model before executing the bounded search, as suggested in Section 4.2.5.

Based on our experience with the two tools, and based on their strengths and limitations (discussed in Sections 4.1.3 and 4.2.5), we have reached several conclusions:

• While testing is an easy-to-use verification approach, it is very time consuming and error prone. For example, testing did not uncover the two bugs that were
uncovered by the verification prototype described in Section 4.2 since evaluation of the output was done manually. Thus, we claim that testing is useful for intermediate verification of transformations while they are being developed. However, more reliable techniques are needed to verify the final transformation.

- Although we demonstrated that the verification prototype (Section 4.2) is practical to use, it suffered from a few non-trivial limitations. First, the bounded search approach does not fully guarantee correctness of the transformation. Second, the prototype is less efficient as we increase the scope. Finally, the translations done between the different formalizations manipulated by the prototype cannot be proven to be sound and complete.

Based on the above mentioned points, we claim that static, formal verification approaches are more rigorous and thus suitable for the analysis of transformations used for the development of safety-critical software. To increase the confidence in the verification results, these approaches need to be unbounded, scalable, and the soundness and completeness of the approach needs to be proved. In Chapter 5, we introduce a static, formal verification tool that we enhanced and extended to address the limitations of existing tools (such as the two tools experimented with in this Chapter) by performing unbounded verification of many property types on a sound and complete representation of transformation executions. Moreover, we demonstrate the scalability and applicability of the tool we describe in Chapter 5 by conducting two case studies that we discuss in detail in Chapter 6.
### Multiplicity Invariants:

- **(M1)** Context CompositionType inv CompositionType.component: 
  \[ \text{size}() \geq 1 \]

- **(M2)** Context SoftwareComposition inv SoftwareComposition.softwareComposition: 
  \[ \text{size}() \neq \text{null} \]

- **(M3)** Context SwcToEcuMapping inv SwcToEcuMapping.component: 
  \[ \text{size}() \geq 1 \]

- **(M4)** Context SwcToEcuMapping inv SwcToEcuMapping.ecuInstance: 
  \[ \text{size}() \neq \text{null} \]

- **(M5)** Context System inv System.softwareComposition: 
  \[ \text{size}() \neq \text{null} \]

- **(M6)** Context System inv System.mapping: 
  \[ \text{size}() \neq \text{null} \]

### Uniqueness Contracts: Let Unique (invName, X, Y) be

**Context Global inv**

- **(U1)** UnqCompName= Unique (UNQCOMPNAME, Module, ComponentPrototype)
- **(U2)** UnqSysMName= Unique (UNQSYSMNAME, PhysicalNode, SystemMapping)
- **(U3)** UnqSysName= Unique (UNQSYSNAME, PhysicalNode, System)
- **(U4)** UnqSwcmpsName= Unique (UNQSWMPSNAME, PhysicalNode, SoftwareComposition)
- **(U5)** UnqCmpstyName= Unique (UNQCMPSTYNAME, PhysicalNode, CompositionType)
- **(U6)** UnqEcuInName= Unique (UNQECUINAME, PhysicalNode, EcuInstance)
- **(U7)** UnqS2EName= Unique (UNQS2ENAME, Partition, SwcToEcuMapping)
- **(U8)** UnqPpName= Unique (UNQPPNAME, ExecFrame, PPortPrototype)
- **(U9)** UnqRpName= Unique (UNQRPNAME, ExecFrame, RPortPrototype)

### Security Invariant:

- **(S1)** Context System inv Self, Cont: 
  \[ \text{size}() \neq \text{null} \]

### Pattern Contracts:

- **(P1)** Context Global inv Sig2P: 
  \[ \text{size}() \neq \text{null} \]

- **(P2)** Context Global inv Sig2R: 
  \[ \text{size}() \neq \text{null} \]

### Table 4.2: Formulated OCL Constraints
Chapter 5

The Symbolic Model Transformation Property Prover

After a thorough review of the studies investigating model transformation verification approaches (Chapter 2) and after experimenting with existing transformation verification tools (Chapter 4) on our GM-to-AUTOSAR transformation (Chapter 3), we found several limitations in the state of the art, as explained in Chapter 4. For example, several studies translated either the transformation of interest (e.g., [35, 36]) or the property of interest (e.g., [61]) to an intermediate formalism to facilitate verification, without proving the soundness of the translation and without translating the verification result back to the original formalism for comprehension. Second, other studies proposed incomplete verification techniques that do not account for all possible transformation executions (e.g., verification within a scope [35] and transformation testing [53]). Finally, a large number of studies proposed input-dependent [137, 5] verification techniques (e.g., Henshin [10], AGG [148]) that prove properties for transformations only when run on a specific input. More general, input-independent verification techniques are needed where property verification is to be performed only
once for the model transformation, and verification results are to be guaranteed for all possible inputs.

In collaboration with Levi Lúcio and Bentley J. Oakes from Mc Gill University (Canada), we attempt to overcome limitations of previous studies by investigating verifying properties of transformations implemented in the graph-based model transformation language DSLTrans [18]. We focus on properties expressed as input-output model relations since they have been highly investigated in the literature, using both textual (e.g., [35, 6]) and graphical (e.g., [13, 148]) property languages. DSLTrans is a non-Turing complete transformation language, i.e., DSLTrans cannot specify transformations that require unbounded loops (e.g., simulation transformations). We choose to verify properties for DSLTrans for three main reasons:

- Graph-based model transformation languages are intuitive and do not require a mathematical background to be used, unlike verification approaches (e.g. [152, 106]) that use formalizations such as Maude (Appendix A.5).

- There is a trade off between the expressiveness of model transformation languages and the verifiability of these languages. The limited expressiveness of DSLTrans makes it possible to perform unbounded, formal verification of transformation properties, as we will show in this chapter and in Chapter 6. Albeit the limited expressiveness of DSLTrans, Sections 6.1 and 6.2 demonstrate that DSLTrans can be used to implement non-trivial transformations.

- We develop the verification approach in collaboration with the developers of DSLTrans. This facilitated understanding the inner workings of the DSLTrans language and developing a verification approach for it. In Section 5.2, we explicitly state which component of the verification approach we developed and
which component was developed by our collaborators.

Together with our collaborators, we extend and enhance the features of a symbolic model transformation property prover for DSLTrans that was previously proposed and implemented by Lúcio et al. [90]. The first version of the property prover generated the set of path conditions (i.e., symbolic transformation executions) for an input transformation, and verified atomic contracts (i.e., constraints on input-output model relations) on these path conditions. The original prover evaluated atomic contracts to yield either true or false for the transformation when run on any input model. This first version of the property prover was implemented in Prolog and early scalability tests showed that the property prover did not scale well. Moreover, this prototype was not complete, i.e., it lacked the necessary algorithms to build path conditions representing all transformation executions. The improved property prover described in this chapter was extended in three ways:

- The extended prover supports a more expressive property language that facilitates verifying atomic contracts and compositions of atomic contracts in the form of propositional logic formulae. We formally define the syntax and semantics of the extended property language, and the evaluation of properties expressed in this language. Besides, the extended property prover can handle overlapping rules during path condition generation (Section 5.3).

- To overcome the scalability issue of the first implementation of the property prover and to facilitate verifying properties in the improved property language, the property prover was redeveloped from scratch using Python and T-Core [146].

- Our collaborators proved that the path conditions generated by the extended
property prover are sound and complete [91].

The original property prover [90] and the extended property prover that we present in this chapter are both input-independent [137, 5], i.e., verification results generated by the prover hold for all possible inputs.

The rest of this chapter is organized as follows: Section 5.1 summarizes DSLTrans and its simplest properties; Section 5.2 overviews the architecture of our property prover; Section 5.3 describes the first phase carried out by our property prover (i.e., path condition generation); Section 5.4 describes the second phase carried out by our property prover (i.e., property verification); Section 5.5 summarizes this chapter.

5.1 The DSLTrans Model Transformation Language

DSLTrans [18] is a graph-based model transformation language that can be used to specify out-place (i.e., input-preserving) model transformations that are confluent and terminating by construction. DSLTrans rules are constructive, i.e., elements can be created but not deleted. The semantics of DSLTrans (currently defined using set theory) is in-line with, and can be defined using, pushout approaches (Appendix A.1). We demonstrate DSLTrans using a simple transformation as a running example.

Figs. 5.1 and 5.2 present two metamodels used to describe different representations of a set of people. The ‘Household Language’ represents people as members of families that in turn form a set of households. Each family and each member has a Name attribute to refer to the family name and the member name, respectively. The ‘Community Language’ represents people as men or women who belong to a community. Each person (i.e., man or woman) has a Name attribute to refer to the person’s full name.
Fig. 5.3 presents a DSLTrans transformation that transforms members of a family in the ‘Household Language’ (source metamodel) into men and women of a community in the ‘Community Language’ (target metamodel). In what follows, we refer to the transformation in Fig. 5.3 as the Persons transformation.

A DSLTrans model transformation is composed of an ordered set of layers (e.g., ‘TopLevel’, ‘FamilyMembersToGender’, and ‘BuildCommunityOfPersons’ layers in Fig. 5.3) that are executed sequentially. A layer consists of a set of transformation rules that execute in a non-deterministic order but produce a deterministic result. Each rule is a pair \( (\text{MatchModel}, \text{ApplyModel}) \) where \( \text{MatchModel} \) is a pattern of source metamodel elements and \( \text{ApplyModel} \) is a pattern of target metamodel elements. For example, the MatchModel of the ‘HouseholdsToCommunity’ transformation rule in the ‘TopLevel’ layer (Fig. 5.3) has one ‘Households’ class from the ‘Household Language’ and the ApplyModel has one ‘Community’ class from the ‘Community Language’. This means that input model elements of type ‘Households’ (from the ‘Household Language’) will be transformed into output model elements of type ‘Community’ (from the ‘Community Language’).
Figure 5.3: The *Persons* Transformation expressed in DSLTrans.
5.1. THE DSLTRANS MODEL TRANSFORMATION LANGUAGE

When a DSLTrans rule executes, *traceability links* are created between each element in the rule’s MatchModel and each element in the rule’s ApplyModel. These traceability links are used to keep track of which output model elements came from which input model elements.

We describe some DSLTrans constructs that are used to build the MatchModel of a DSLTrans rule. More DSLTrans constructs can be found in [18, 91].

- **Match Elements** are variables in the MatchModel of a DSLTrans rule that are typed by source metamodel classes and can assume as values instances of that class from the input model. An example of a match element is the ‘Family’ element in the MatchModel of the ‘FatherToMan’ rule in the ‘FamilyMembersToGender’ layer (Fig. 5.3). Match elements can be of two types: *Any* match elements are bound to all matching instances in the input model, and *Exists* match elements are bound to only one (deterministic) matching instance in the input model. All match elements in Fig. 5.3 are of type *Any*.

- **Attribute Conditions** are conditions on the attributes of a match element.

- **Direct Match Links** are links between two match elements that are typed by labelled relations of the source metamodel. These links can assume as values links having the same label in the input model.

- **Indirect Match Links** represent a path of containment associations between two linked match elements. For example, an indirect match link appears in the ‘BuildCommunity’ rule of the ‘BuildCommunityOfPersons’ layer as a horizontal, dashed arrow between the ‘Households’ and ‘Member’ match elements.

Similar constructs can be used to build the ApplyModel of a DSLTrans rule, as shown in Fig. 5.3.
• *Apply elements* are variables in the ApplyModel of a DSLTrans rule that are typed by target metamodel classes and linked by *apply links*. Apply elements that are not connected by backward links create output elements of the same type each time the MatchModel is found in the input. Apply elements that are connected by backward links are handled differently, e.g., ‘BuildCommunity’ rule of the ‘BuildCommunityOf-Persons’ layer connects ‘Community’ and ‘Person’ output elements that were formerly created from ‘Households’ and ‘Member’ input elements with a ‘has’ link.

• Apply elements can have *apply attributes* that can be set from references to one or more attributes of match elements. For example, the ‘FatherToMan’ rule of the ‘LayerFamilyMembersToGender’ layer sets the *Name* apply attribute of a ‘Man’ apply element to be equal to the concatenation of the *Name* match attributes of the corresponding ‘Member’ and ‘Family’ match elements.

• *Backward Links* link elements of the ApplyModel and the MatchModel of a rule, e.g., backward links are used in the ‘BuildCommunity’ rule of the ‘BuildCommunityOf-Persons’ layer and are denoted as vertical, dashed lines. Backward links are used to refer to traceability links between input and output model elements that have been already generated by the rules of previous layers.

• *Free variables* can occur in any element $e$ of an rule’s ApplyModel and can be used with backward links. The first occurrence of a free variable (without a backward link) binds the variable to element $e$ generated by the rule. Any successive occurrences of the free variable (with backward links) matches only previously generated elements that have been bound to the same free variable. In other words, using the same free variable in different rules together with backward links allows these rules to refer to the same generated element. For example, the ‘HouseholdsToCommunity’ rule (‘TopLevel’ layer) binds the generated ‘Community’ element to the free variable
‘COMM’ such that this element can be referred to in the ‘BuildCommunity’ rule (‘BuildCommunityOfPersons’ layer) by using the same free variable ‘COMM’ and a backward link in the ‘BuildCommunity’ rule. The first occurrence of a free variable (without a backward link) must be in a layer preceding the layer where successive occurrences of the same free variable (with backward links) appear. However, a free variable can occur for the first time (without a backward link) in several rules of the same layer.

**AtomicContracts in DSLTrans:** An *AtomicContract* is the simplest DSLTrans property that can be expressed in our prover. Each *AtomicContract* is a pair (*pre*, *post*) that specifies a property of the form: “if the input model satisfies the precondition *pre*, then the output model should satisfy the postcondition *post*”. A precondition is a constraint on the input model of the transformation in the form of a structural relation between input model elements. A postcondition is a constraint on the output model of the transformation in the form of a structural relation between output model elements. Preconditions and postconditions are expressed using the same constructs as rules (described above). Postconditions may also have traceability links to link postcondition elements to precondition elements. Traceability links in postconditions signify that the property will only match an output model element that was previously created from (and hence, linked to) the input model element.

Figs. 5.4 and 5.5 demonstrate two *AtomicContracts* for the *Persons* transformation (Fig. 5.3). The *AtomicContract* shown in Fig. 5.4 is interpreted as: “a mother and a father in a family will always be transformed to a woman and a man”. The *AtomicContract* shown in Fig. 5.5 is interpreted as: “a family including a mother and a daughter will always be transformed to a man”. Our property prover should
\[\text{Precondition}\]
\[\text{Member}\]
\[\text{Family}\]
\[\text{Postcondition}\]
\[\text{Man}\]
\[\text{Woman}\]
\[\text{Member}\]

\[\text{Contract}\] 1

\[\text{Precondition}\]
\[\text{Member}\]
\[\text{Family}\]
\[\text{Postcondition}\]
\[\text{Man}\]
\[\text{Member}\]

\[\text{Contract}\] 2

\[\text{Precondition}\]
\[\text{Member}\]
\[\text{Family}\]
\[\text{Postcondition}\]
\[\text{Man}\]

\[\text{Contract1}\]; should hold.

\[\text{Contract2}\]; should not hold.

verify that \textit{Contract1} (Fig. 5.4) will always hold for the \textit{Persons} transformation, and \textit{Contract2} (Fig. 5.5) will not always hold (with a counterexample). \textit{AtomicContracts} are formally defined in [91].

\section*{5.2 Overview of the Symbolic Model Transformation Property Prover}

Fig. 5.6 demonstrates the final architecture of our symbolic model transformation property prover. Our property prover takes four inputs: the DSLTrans transformation of interest, the source and target metamodels manipulated by the transformation, and the property to verify. Verification is then carried out in two steps, as shown in Fig. 5.6. First, the property prover generates the set of \textit{path conditions} representing all possible executions of the input transformation (Section 5.3). Then, the prover verifies the input property on the generated set of path conditions and renders the property to be either \textit{true} or \textit{false} for the input transformation when run on any input model (Section 5.4). If the input property is \textit{false} for the input transformation, a counterexample is generated. The counter example is comprised of a path condition for which the property does not hold. Thus, according to the classification we presented.
5.2. OVERVIEW OF THE SYMBOLIC MODEL TRANSFORMATION PROPERTY PROVER

in [137, 5, 3], the verification technique used by our property prover is transformation-dependent and input-independent.

When deciding on a transformation language to implement our symbolic model transformation property prover, the following aspects were taken into consideration: (1) The steps carried out by our property prover require graph manipulation. (2) Support for higher order transformations (HOTs) is necessary to automate the implementation of our property prover. Thus, a model transformation language with an explicit metamodel is required. (3) A model transformation language with a control flow mechanism is needed to allow scheduling HOT rules.

After considering the former points, we have chosen to implement our symbolic model transformation property prover using Python and the T-Core framework [146]. T-Core is a Python library with primitives that support typed graph manipulation (e.g., graph matching, graph rewriting, rule backtracking, iterating, and resolving potential conflicts between matchings and rewritings) and composition of these primitives into transformation blocks. The use of Python and T-Core allowed developing our property prover using MDD principles. In other words, all artifacts used at verification run-time are models (i.e., instances of explicit metamodels), all model-related computations are implemented as model transformations, and all computations that do not directly manipulate models are implemented as Python algorithms that have been optimized to minimize memory usage and run-time. The models, metamodels, and model transformations used at verification run-time are themselves automatically
5.3. PHASE I: GENERATING THE SET OF PATH CONDITIONS

Our property prover generates a set of *path conditions* that symbolically represent the possible executions of the input model transformation. For a model transformation with *n* layers, our property prover uses the model transformation rules to build the path conditions in *n* iterations. Fig. 5.7 demonstrates how the path conditions for the
5.3. PHASE I: GENERATING THE SET OF PATH CONDITIONS

Persons transformation (Fig. 5.3) are generated in iterations $^1$. Every rule in each layer of Fig. 5.3 is identified with a pair of numbers, e.g., $4_2$ corresponds to the fourth rule (ordered from top to bottom and then from left to right in Fig. 5.3) in the second layer (i.e., ‘SonToMan’ rule in the Persons transformation). We start off with the empty path condition, where we assume no transformation rule has been applied. To generate path conditions in iteration 1, the empty path condition is combined with all possible rule combinations of the first layer in the Persons transformation. Similarly, to generate path conditions in iteration 2, each path condition from iteration 1 is combined with all applicable rule combinations of the second layer in the Persons transformation. A rule combination of the second layer that does not have backward links is always applicable, since it does not depend on rules from the first layer. Rule combinations of the second layer with backward links are combined with a path condition from iteration 1 only if the path condition generates the elements linked by backward links in the rule combination of the second layer.

Therefore, the path conditions generated in iteration $i$ include the power set of

$^1$Since every node in the execution tree (shown in Fig. 5.7) holds a (partial) path condition, we use the terms ‘node’ and ‘path condition’ interchangeably.
5.3. PHASE I: GENERATING THE SET OF PATH CONDITIONS

rules from the $i^{th}$ transformation layer and rules from the path conditions generated until iteration $i$-1. Each path condition thus accumulates a set of rules describing a possible path of rule applications through the layers of the model transformation. The accumulated MatchModels of all the rules in a path condition are referred to as the path condition’s *match pattern*. Similarly, the accumulated ApplyModels of all the rules in a path condition are referred to as the path condition’s *apply pattern*. Since the path condition generation technique abstracts from how many times the rule executes for an input, a transformation rule only occurs once in each path condition. Thus, a path condition symbolically represents a set of concrete executions since each of the rules in a path condition can be concretely executed any number of times on an input model.

Fig. 5.8 shows the path condition of the node with the dashed edge in Fig. 5.7. As shown from the numbers in the node, the path condition contains four combined rules from the *Persons* transformation (i.e., ‘HouseholdsToCommunity’, ‘FatherToMan’, ‘MotherToWoman’, ‘BuildCommunity’), their attribute manipulation, and their traceability links. When combining the rules, elements of the same type of the combined rules can be merged. This represents the fact that different rules may execute over the same input elements. Common match links, apply elements, and apply links are accumulated similarly.

Only the path conditions from the last iteration are returned as the result since they capture all the possible *complete* model transformation executions. For a model transformation $\tau$, we refer to this resulting set of path conditions as $PC_{\tau}$, i.e., $PC_{\tau} = \{pc_i \mid pc_i$ is a path condition containing a possible sequence of rule applications in $\tau\}$. Details on the improved path condition generation algorithm can be found in [91].
5.3. PHASE I: GENERATING THE SET OF PATH CONDITIONS

Figure 5.8: A path condition containing four combined rules from the Persons transformation (i.e., ‘HouseholdsToCommunity’, ‘FatherToMan’, ‘MotherToWoman’, ‘BuildCommunity’).

Overlapping Rules: The DSLTrans version of our GM-to-AUTOSAR transformation presented later in Section 6.1 had overlapping rules which required treatment during path condition generation. Overlapping rules are defined as follows: when two rules in the same layer use match elements of the same metamodel classes of type Any match class or Exists match class (described in Section 5.1), then the MatchModel of one rule syntactically subsumes the MatchModel of the other rule. For example, a rule having a MatchModel containing an Any match element of class ‘A’ is subsumed by a MatchModel of another rule that contains an Exists match element of class ‘A’ and an Any match element of class ‘B’.

The path condition generation algorithm was extended to handle overlapping rules. Depending on whether rules overlap totally or partially, rule merge may be
done before or during path condition generation. For transformations with rule overlaps, this extension leads to an improved management of the combinatorial explosion in path condition generation, i.e., there is a pronounced decrease in the number of generated path conditions, since rules in a subsumption relation (described above) can often be merged into a smaller set of rules. Handling overlapping rules during path condition generation is explained in detail in [91].

5.4 Phase II: Verification of the Property of Interest

We extend the verification approach proposed in [90, 93] for verifying AtomicContracts of DSLTrans model transformations to enable the verification of more complex properties. Our extended technique employs the following syntax and semantics.

**Syntax:** Our syntax is based on propositional logic [66]. An AtomicContract \((pre, post)\) is the smallest unit in our property language (described in Section 5.1). A propositional formula can be built using one or more AtomicContracts and the operators \(\neg_{tc}\) (not), \(\lor_{tc}\) (or), \(\land_{tc}\) (and), and \(\Rightarrow_{tc}\) (implication), where \(tc\) stands for “transformation contract”. Assuming that \((pre, post)\) is an element of the set of AtomicContracts \(AC\), the syntax of propositional formulae that are expressible in our property prover is:

\[
\varphi := (pre, post) | \neg_{tc}\varphi | \varphi \lor_{tc} \varphi | \varphi \land_{tc} \varphi | \varphi \Rightarrow_{tc} \varphi \tag{5.1}
\]

*Free variables* can occur in any element \(e\) of an AtomicContract’s precondition or postcondition. This occurrence binds the free variable to all the matches found for \(e\) within an instantiation of a MatchModel. Using the same free variable in different AtomicContracts allows these AtomicContracts to refer to the same matched element. For example, AtomicContract cont1 in Fig. 5.9 binds a matched element
of type ‘Community’ to the free variable ‘COMMUNITY’ such that this element can be referred to in AtomicContracts cont2 and cont3. The bindings of a set of free variables \( \{ \text{var}_1, \ldots, \text{var}_n \} \) (occurring in elements \( \{ e_1, \ldots, e_i \} \) of an AtomicContract) to matched elements \( \{ m_1, \ldots, m_n \} \) in a path condition are expressed as a binding function \( l = \{(\text{var}_1, m_1), \ldots, (\text{var}_n, m_n)\} \), i.e., \( l \in FV \mapsto BE \), where \( FV \) is the the set of free variables and \( BE \) is the set of bound elements\(^2\).

**Semantics:** We define a function \( \text{eval}_{\text{Atomic}}(pc, c) \) that evaluates an AtomicContract \( c = (pre, post) \) for a path condition \( pc \) as follows:

1. If \( pc \) contains an isomorphic copy of \( pre \) but does not contain an isomorphic copy of \( post \), then \( \text{eval}_{\text{Atomic}}(pc, c) \) returns \( false \) (i.e., \( c \) does not hold for \( pc \) and hence, does not hold for the transformation) and an empty set of binding functions \( L = \emptyset \).

2. If \( pc \) contains an isomorphic copy of \( pre \) and an isomorphic copy of \( post \), \( \text{eval}_{\text{Atomic}}(pc, c) \) returns \( true \) (i.e., \( c \) holds for \( pc \)) and a set of binding functions \( L \) for the free variables of \( c \), where \( L \in \mathcal{P}(FV \mapsto BE) \) and \( \mathcal{P} \) is the power set operator.

3. If \( pc \) does not contain an isomorphic copy of \( pre \), \( \text{eval}_{\text{Atomic}}(pc, c) \) returns \( true \) (i.e., \( c \) holds for \( pc \)) and an empty set of binding functions \( L = \emptyset \).

Thus, \( \text{eval}_{\text{Atomic}} \) is defined as \( \text{eval}_{\text{Atomic}} : PC_\tau \times AC \rightarrow \{true, false\} \times \mathcal{P}(FV \mapsto BE) \), where \( PC_\tau \) is the set of path conditions of a model transformation \( \tau \) (as described in Section 5.3). Note that a set \( L \) of binding functions is returned since an AtomicContract may evaluate to true using different bindings of the free variables.

Thus, the set \( L \) is constructed from all binding functions \( l_i \) returned by all possible

\(^2\)To avoid the confusion, we emphasize that the purpose of using free variables in AtomicContracts is similar to the purpose of using free variables in transformation rules (as described in Section 5.1). Using the same free variable in different transformation rules (together with backward links) allows these rules to refer to the same element. Similarly, using the same free variable in different AtomicContracts allows these AtomicContracts to refer to the same element.
subgraph isomorphisms.

Assuming that \( FORMULAE \) is the set of elements generated by the grammar in Eqn.(5.1), we evaluate a formula \( \varphi \) for a path condition \( pc \in PC_\tau \) using a function \( eval : PC_\tau \times FORMULAE \rightarrow \{true, false\} \times P(FV \not\rightarrow BE) \) as follows:

\[
eval(pc, \varphi) = \begin{cases} 
(res_1, L_1) & \text{if } \varphi \in AC, 
\eval\text{Atomic}(pc, \varphi) = (res_1, L_1) \\
(-res_1, L_1) & \text{if } \varphi = \neg tc \psi, 
\eval(pc, \psi) = (res_1, L_1) \\
((res_1 \lor res_2) \land C(L_1, L_2), 
\ eval(pc, \phi) = (res_2, L_2) & \text{if } \varphi = \psi \lor tc \phi, 
\eval(pc, \psi) = (res_1, L_1), \\
L_1 \cup L_2) & \text{eval}\text{Atomic}(pc, \varphi) = (res_1, L_1) \\
((res_1 \land res_2) \land C(L_1, L_2), 
\ eval(pc, \phi) = (res_2, L_2) & \text{if } \varphi = \psi \land tc \phi, 
\eval(pc, \psi) = (res_1, L_1), \\
L_1 \cup L_2) & \text{eval}\text{Atomic}(pc, \varphi) = (res_1, L_1) \\
((res_1 \implies res_2) \land C(L_1, L_2), 
\ eval(pc, \phi) = (res_2, L_2) & \text{if } \varphi = \psi \implies tc \phi, 
\eval(pc, \psi) = (res_1, L_1), \\
L_1 \cup L_2) & \text{eval}\text{Atomic}(pc, \varphi) = (res_1, L_1) \\
\end{cases}
\]

(5.2)

where the semantics of the propositional operators \( (\neg tc, \lor tc, \land tc, \implies tc) \) is standard, and \( res_i \in \{true, false\} \). The consistency function \( C : P(FV \not\rightarrow BE) \times P(FV \not\rightarrow BE) \rightarrow \{true, false\} \) checks for two sets of binding functions (e.g., \( L \) and \( L' \)) that all free variables bound by a binding function \( l_i \) in the first set \( L \) will always be bound to the same elements by a binding function \( l'_j \) of the second set \( L' \), i.e., given two sets of binding functions \( L \) and \( L' \) we have:

\[
C(L, L') = \forall l \in L, \exists l' \in L' : (\forall v \in FV_l : ((v, m) \in l \land (v, m') \in l') \implies m = m') \text{ and } \\
\forall l' \in L', \exists l \in L : (\forall v \in FV_{l'} : ((v, m') \in l' \land (v, m) \in l) \implies m' = m)
\]

(5.3)

where \( m, m' \in BE \), and \( FV_l, FV_{l'} \) are the sets of free variables used in \( l \) and \( l' \), respectively. Based on the former definitions, we evaluate a formula \( \varphi \) for a transformation
\( \tau \) (with path conditions \( PC_\tau \)) using a function \( \text{eval}(\tau, \varphi) \) defined as follows:

\[
\text{eval}(\tau, \varphi) = \begin{cases} 
\text{true} & \text{if } \forall pc \in PC_\tau : \text{eval}(pc, \varphi) = (\text{true}, L) \\
\text{false} & \text{otherwise}
\end{cases}
\] (5.4)

where \( L \) is any set of binding functions. Thus, \( \text{eval}(\tau, \varphi) \) renders a property \( \varphi \) to be \( \text{true} \) or \( \text{false} \) for a model transformation \( \tau \) by verifying \( \varphi \) for each path condition in the generated path conditions. Function \( \text{eval}(\tau, \varphi) \) returns \( \text{true} \) only if for all path conditions of \( \tau \), \( \varphi \) holds and the bindings of all free variables are consistent (as described above in Eqn. 5.3).

**Formulae of AtomicContracts:** The new syntax and semantics allows us to formulate complex properties by composing propositional formulae of AtomicContracts. We demonstrate how the AtomicContracts in Fig. 5.9 (i.e., \( \text{cont1}, \text{cont2}, \) and \( \text{cont3} \)) together with free variables can be used with different propositional operators to convey different multiplicity invariants \(^3\). A property that mandates that the Persons transformation (Figure 5.3) will always generate an output model where every community has one or more elements of type ‘Person’ (i.e., a multiplicity invariant of ‘1..\(^*\)’) can be expressed as ‘\( \text{cont1} \implies_{tc} \text{cont2} \)’. In other words, if an element of type ‘Community’ is generated in the output model, then this ‘Community’ element (as per the free variable ‘COMMUNITY’) must have at least one element of type ‘Person’. Whereas the property ‘\( \text{cont1} \implies_{tc} (\text{cont2} \land_{tc} \neg_{tc} \text{cont3}) \)’ expresses a multiplicity invariant of ‘1..1’ (i.e, if an element of type ‘Community’ is generated in the output model, then this ‘Community’ element must have one element of type ‘Person’, and not more than one).

We implemented our symbolic model transformation property prover such that it

\(^3\)Note that the three AtomicContracts in Fig. 5.9 have empty preconditions meaning that they will match on any input model.
5.5. SUMMARY

Figure 5.9: Three AtomicContracts (i.e., cont1, cont2, and cont3) that can be used with different propositional operators to convey different properties for the Persons model transformation.

follows the semantics described above closely. Similar to any other formal verification approach, we cannot prove the equivalence of the semantics and the implemented prover. However, we have closely examined the implementation and tested it with several properties to ensure that our property prover reflects the intended semantics.

5.5 Summary

In this chapter, we presented an input-independent, symbolic model transformation property prover for the DSLTrans model transformation language. The property prover is intended to address the limitations of the state of the art verification tools, such as those we experimented with in Chapter 4. Lúcio et al. [90] initially proposed and implemented a Prolog prototype of the property prover. In collaboration with Levi Lúcio and Bentley J. Oakes from Mc Gill University (Canada), we redevelop the property prover using Python and T-core to improve the scalability of the prover and to allow the verification of more complex properties. Our collaborators from McGill University were responsible for redeveloping the first phase of our symbolic model
transformation property prover (i.e., the generation of the path conditions) [94, 91, 93] and we were responsible for redeveloping the second phase of our property prover (i.e., property verification using the generated path conditions). The soundness and completeness of the generated path conditions were proved by our collaborators [91].

We introduced the basics of DSLTrans and the smallest provable property in DSLTrans (i.e., AtomicContract) in Section 5.1. Then, we demonstrated the overall architecture of our property prover in Section 5.2. Finally, we discussed the two steps carried out by our property prover in Sections 5.3 and 5.4.
Chapter 6

Evaluation of the Symbolic Model Transformation Property Prover

In this chapter, we conduct two case studies to evaluate our symbolic model transformation property prover that we discussed in Chapter 5. We present the first case study in Section 6.1, where we use our symbolic model transformation property prover to verify the GM-to-AUTOSAR model transformation (described in Chapter 3). We present the second case study in Section 6.2, where we use our symbolic model transformation property prover to verify a model transformation that maps UML-RT state machines to Kiltera process models [120]. For each case study, we discuss the objective of the model transformation, the required mapping between the source and target metamodels, the implementation of the model transformation in DSLTrans, the properties of interest for the model transformation, the formulation of these properties in our property prover, the verification results, the performance of our property prover, and how our property prover helped uncover and fix any bugs in the model transformation. Based on the two case studies, we discuss the strengths and limitations of our property prover in Section 6.3 and we summarize this chapter...
6.1. THE GM-TO-AUTOSAR CASE STUDY

in Section 6.4.

We specifically choose these two case studies to evaluate our symbolic model transformation property prover for several reasons. First, the two model transformations used as case studies cover a wide spectrum of model transformations of varying sizes. For example, the GM-to-AUTOSAR model transformation is an industrial, model-to-model transformation of structural models. Whereas the UML-RT-to-Kiltera model transformation is an academic, model-to-text transformation of behavioural models. The two model transformations also differ in their intents [4, 89]: the GM-to-AUTOSAR model transformation is a migration transformation, while the UML-RT-to-Kiltera model transformation is an analysis transformation. Second, verification of the GM-to-AUTOSAR model transformation is of relevance to our industrial partners. Third, the two model transformations were developed by our research group and hence, are available for our use. Finally, the two model transformations were not custom-built to test our property prover. Instead, they were developed for different projects and were reused to evaluate our property prover. Using already existing model transformations for evaluation (as opposed to using model transformations created specifically to evaluate our property prover) is intentional to understand how usable and applicable is our property prover on existing model transformations.

6.1 The GM-to-AUTOSAR Case Study

In Chapter 3, we reported an industrial model transformation that maps between subsets of a legacy metamodel for General Motors (GM) and the AUTOSAR metamodel. Specifically, we focused on subsets of the metamodels that represent the deployment
and interaction of software components. Later in Chapter 4 (Section 4.2.3), we proposed properties of interest for our GM-to-AUTOSAR model transformation.

For our first case study, we use our symbolic model transformation property prover to formulate and verify the properties proposed in Section 4.2.3 for the GM-to-AUTOSAR model transformation (Chapter 3) after reimplementing it in DSLTrans. First, we discuss the reimplementation of the GM-to-AUTOSAR model transformation in DSLTrans (Section 6.1.1). Then, we describe the formulation of properties for the GM-to-AUTOSAR model transformation in our property prover (Section 6.1.2). Finally, we report on the verification results of the formulated properties and we elaborate on the performance of our property prover for the GM-to-AUTOSAR model transformation (Section 6.1.3).

6.1.1 Reimplementation of the GM-to-AUTOSAR Model Transformation in DSLTrans

We reimplement the GM-to-AUTOSAR model transformation (described in Chapter 3) in DSLTrans so that we can verify it using our symbolic model transformation property prover. Table 6.1 summarizes the transformation rules in each transformation layer, the input types that are mapped by each transformation rule, and the output types that are generated by each transformation rule. Transformation rules of the first and third transformation layers create output model elements in the transformation’s output model. Transformation rules of the second transformation layer generate associations between output model elements created by transformation rules in the first layer (shown in the actual model transformation using backward links). Thus, the input types and the output types shown for the transformation rules of the
6.1. THE GM-TO-AUTOSAR CASE STUDY

<table>
<thead>
<tr>
<th>Layer</th>
<th>Rule Name</th>
<th>Input Types</th>
<th>Output Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MapPhysNode2FiveElements</td>
<td>PhysicalNode</td>
<td>System, SystemMapping, SoftwareComposition, CompositionType, EcuInstance</td>
</tr>
<tr>
<td></td>
<td>MapPartition</td>
<td>Partition</td>
<td>SwcToEcuMapping</td>
</tr>
<tr>
<td></td>
<td>MapModule</td>
<td>Module</td>
<td>SwCompToEcuMapping_component, ComponentPrototype</td>
</tr>
<tr>
<td>2</td>
<td>MapConnPhysNode2Partition</td>
<td>PhysicalNode, Partition</td>
<td>SystemMapping, EcuInstance, SwcToEcuMapping</td>
</tr>
<tr>
<td></td>
<td>MapConnPartition2Module</td>
<td>PhysicalNode, Partition, Module</td>
<td>CompositionType, ComponentPrototype, SwcToEcuMapping, SwCompToEcuMapping_component</td>
</tr>
<tr>
<td>3</td>
<td>CreatePPortPrototype</td>
<td>ExecFrame</td>
<td>PPortPrototype</td>
</tr>
<tr>
<td></td>
<td>CreateRPortPrototype</td>
<td>ExecFrame</td>
<td>RPortPrototype</td>
</tr>
</tbody>
</table>

Table 6.1: The rules in each layer of the GM-to-AUTOSAR model transformation after reimplementing the transformation in DSLTrans, and their input and output types.

second transformation layer are types that have already been matched and created and for which the transformation rules create associations.

To represent positive application conditions (PACs) in our GM-to-AUTOSAR model transformation rules, we use a combination of Any and Exists match elements (explained in Section 5.1). For example, rule ‘MapPhysNode2FiveElements’ in Table 6.1 maps every PhysicalNode element to five elements, only if the PhysicalNode element is (eventually) connected to at least one Module element. Thus, the MatchModel of rule ‘MapPhysNode2FiveElements’ has a PhysicalNode element (represented as an Any match element) connected to a Partition element and a Module element (represented as Exists match elements). Similarly, rule ‘MapModule’ maps every Module element (represented as Any match element) only if it is contained in one PhysicalNode element and one Partition element (represented as Exists match elements). The MatchModel of rule ‘MapPartition’ also has a Partition element (represented as an Any match element) connected to a PhysicalNode element and a Module element (represented as Exists match elements) to represent a PAC.
Thus, the three rules in the first transformation layer totally overlap (described in Section 5.3) if we abstract from the types of the match elements (i.e., *Any* or *Exists* match elements). The extension to the property prover explained in Section 5.3 combines the three rules of the first transformation layer into one path condition which simplifies property verification. Partially overlapping rules (described in Section 5.3) also occur in the second layer of our GM-to-AUTOSAR model transformation.

The industrial GM-to-AUTOSAR model transformation is not made available for confidentiality reasons.

### 6.1.2 Formulation of Properties for the GM-to-AUTOSAR Model Transformation

In Section 4.2.3, we described properties of interest for the GM-2-AUTOSAR transformation and we summarized them in Table 4.2. In this Section, we demonstrate how we formulate the properties shown in Table 4.2 in our property prover\(^1\).

![Diagram of AtomicContract AC1](image)

**Figure 6.1:** *AtomicContract AC1* that is used to express property *P1*.

We demonstrate the formulation of pattern contracts (e.g., *P1* and *P2* in Table 4.2) in our property prover by showing the formulation of *P1* in Fig. 6.1 as an

\(^1\)We demonstrate the formulation of only a subset of the properties shown in Table 4.2 for reasons of confidentiality. However, the results of verifying all the properties in Table 4.2 will be discussed in Section 6.1.3.
example. \( P1 \) mandates that if a \( \text{PhysicalNode} \) element is connected to a \( \text{Service} \) element through the \( \text{provided} \) association in the input model (as shown in the precondition of Fig. 6.1), then the corresponding \( \text{CompositionType} \) element will be connected to a \( \text{PPortPrototype} \) element in the output model (as shown in the postcondition of Fig. 6.1). As explained in Section 5.1, using a traceability link in Fig. 6.1 mandates that \( P1 \) will only match \( \text{CompositionType} \) elements that were previously created from \( \text{PhysicalNode} \) elements. We demonstrate the formulation of ‘1..1’ multiplicity invariants (e.g., \( M2, M4, M5, \) and \( M6 \) in Table 4.2) by showing the formulation of \( M6 \) in Fig. 6.2 as an example. \( M6 \) ensures that if a \( \text{System} \) element is created in the output model, then this specific \( \text{System} \) element must be connected to one \( \text{SystemMapping} \) element (and not more). Using the \( \text{AtomicContracts} \) shown in Fig. 6.2, \( M6 \) can be expressed as \( AC2 \implies_{tc} (AC3 \land_{tc} \neg_{tc} AC4) \). The free variable ‘SYSTEM’ in Fig. 6.2 mandates that if \( AC2 \) holds for a specific \( \text{System} \) element, then \( AC3 \) should hold and \( AC4 \) should not hold for the same \( \text{System} \) element. Changing the former formula to \( AC2 \implies_{tc} AC3 \) expresses a ‘l..*’ multiplicity invariant (e.g., \( M1, M3 \)). Using the \( \text{AtomicContracts} \) shown in Fig. 6.3, the security invariant \( S1 \) can be expressed as \( AC5 \implies_{tc} AC6 \). The free variables ‘SYSTEM’ and ‘COMPONENTPROTOTYPE’
in Fig. 6.3 mandate that if $AC5$ holds for a specific $System$ element and a specific $ComponentPrototype$ element, then $AC6$ should also hold for the same $System$ element and $ComponentPrototype$ element.

Figure 6.3: *AtomicContracts* $AC5$ and $AC6$ that are used to express property $S1$ as $AC5 \Rightarrow_{tc} AC6$.

### 6.1.3 Verification Results

We used our symbolic model transformation property prover to verify the properties shown in Table 4.2 for our GM-to-AUTOSAR model transformation. The transformation was found to violate the multiplicity invariants $M1$ and $M3$, i.e., our GM-to-AUTOSAR model transformation can generate an output model with the following bugs: (i) A $CompositionType$ element that is not connected to at least one $ComponentPrototype$ element through the $component$ association (violating $M1$), and/or (ii) A $SwcToEcuMapping$ element that is not connected to at least one $SwcToEcuMapping_{component}$ element through the $component$ association (violating $M3$). Thus, our property prover uncovered the same bugs that we found in the ATL transformation implementation using another tool in Chapter 4 (Section 4.2.4). After examining the generated counter examples, we identified and fixed the two bugs in the DSLTrans
implementation of the GM-to-AUTOSAR model transformation. The properties were reverified on the updated GM-to-AUTOSAR model transformation, and verification of all the properties in Table 4.2 returned true, i.e., the DSLTrans implementation of our GM-to-AUTOSAR model transformation will always satisfy the properties shown in Table 4.2.

To assess the performance of our symbolic model transformation property prover, we measured the time taken to generate the path conditions and the time taken to verify the properties (shown in Table 4.2) of the GM-to-AUTOSAR model transformation after fixing the bugs. Our property prover took on average 0.6 seconds to generate the path conditions. The first row of Table 6.2 shows the time taken (in seconds) to verify the properties shown in Table 4.2 using the generated path conditions. We do not include the time taken for path condition generation in the first row of Table 6.2 since it is performed once for the model transformation. The longest time taken to verify a property was 0.02 seconds for verifying $P_1$ and $P_2$. Thus, our property prover can verify properties of an industrial transformation in a short time.

<table>
<thead>
<tr>
<th>Property</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>S1</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification Time (our property prover)</td>
<td>.013</td>
<td>.017</td>
<td>.013</td>
<td>.017</td>
<td>.017</td>
<td>.019</td>
<td>.017</td>
<td>.02</td>
<td>.02</td>
</tr>
<tr>
<td>Verification Time (Section 4.2 at scope 6)</td>
<td>76</td>
<td>73.4</td>
<td>75</td>
<td>75</td>
<td>75.5</td>
<td>74.5</td>
<td>114</td>
<td>250</td>
<td>251</td>
</tr>
</tbody>
</table>

Table 6.2: Time taken (in seconds) to verify the properties in Table 4.2 using our property prover (first row) and using a tool based on model finders that we experimented with in Section 4.2 (second row).

The second row of Table 6.2 shows the time taken (in seconds) to verify the same properties (shown in Table 4.2) using the tool we experimented with in Section 4.2.
The second row of Table 6.2 shows the results for the smallest scope we used in Section 4.2 (i.e., 6). As shown in Table 6.2, our property prover takes a significantly shorter time to exhaustively verify the properties, whereas much longer times were needed to verify the same properties in a scope of 6 in Section 4.2. Thus, we claim that our prover scales well in comparison with the tool we used in Section 4.2. More experiments are needed before we can claim that our property prover scales to industrial model transformations of varying complexities.

Our symbolic model transformation property prover and the model transformation used in [18, 93] are available on-line at [92].

6.2 The UMLRT-to-Kiltera Case Study

Posse and Dingel [120] reported on a model transformation that maps UML-RT [133, 135, 134] models to Kiltera [116, 117, 118, 119, 121] process models. Specifically, the study focused on transforming only behavioral state machine diagrams and structure diagrams (or capsule diagrams) of UML-RT, since these are the most important for UML modelling [120]. Later, Paen [109] implemented the mapping proposed by Posse and Dingel [120] as a model transformation in ATL, focusing only on transforming UML-RT state machine diagrams (less the history states and the enabled transition selection policy). We refer to this model transformation as the UML-RT-to-Kiltera model transformation.

For our second case study, we use our symbolic model transformation property prover to formulate and verify properties of interest for a DSLTrans implementation of the UML-RT-to-Kiltera model transformation [120]. In Section 6.2.1, we describe the model transformation problem, including the source and target metamodels. In
Section 6.2.2, we summarize the mapping proposed by Posse and Dingel [120] between the manipulated metamodels. In Section 6.2.3, we discuss the reimplementation of the ATL UML-RT-to-Kiltera model transformation (originally developed by Paen [109]) in DSLTrans. Then, we describe properties of interest for the UML-RT-to-Kiltera model transformation in Section 6.2.4 and the formulation of these properties in our property prover in Section 6.2.5. Finally, we demonstrate the verification results and we elaborate on the performance of our property prover for the UML-RT-to-Kiltera model transformation in Section 6.2.6.

6.2.1 The Model Transformation Problem Description

To analyze models in any modelling language, the modelling language must have well defined semantics to avoid having ambiguous models and analysis results. UML-RT [133, 135, 134] is a modelling language with widespread use in many industrial sectors (e.g., automotive, avionics, and telecommunications [120]). However, the semantics of UML-RT has been only defined informally, making the analysis of existing UML-RT models not feasible in an automated manner.

To address this issue, many studies proposed a transformation from UML-RT models to other languages with well-defined semantics, so that the UML-RT models can be analyzed automatically (e.g., [43, 44, 153, 52, 85, 156]). However, these studies either considered a limited subset of the language or made unrealistic assumptions about the UML-RT models that do not necessarily hold, and that can produce erroneous analysis results [120].

To overcome limitations of the aforementioned studies, Posse and Dingel [120]
proposed a more comprehensive mapping or transformation from UML-RT behavioral state machine diagrams and structure diagrams (or capsule diagrams) into Kiltera [116, 117, 118, 119, 121] process models. Kiltera is a modelling language with well-defined, formal semantics and with modelling concepts similar to those used in UML-RT. Paen [109] implemented the mapping of only UML-RT state machines into Kiltera process models [120] (less the history states and the enabled transition selection policy) as an ATL model transformation. We refer to this model transformation as the UML-RT-to-Kiltera model transformation. The UML-RT-to-Kiltera model transformation is intended to support the execution and analysis of UML-RT models.

We discuss in what follows the source UML-RT metamodel and the target Kiltera metamodel of the UML-RT-to-Kiltera model transformation [120, 109].

The Source UML-RT Metamodel

UML-RT [133, 135, 134] is a UML profile for modelling the structure and behavior of event-driven, soft real-time embedded systems. The structure of such systems is specified as a structure diagram or a capsule diagram that consists of a set of system components or capsules. The behavior of these capsules (and hence the system’s behavior) is specified using state machine diagrams.

The complete UML-RT metamodel considered by Posse and Dingel [120] and manipulated by Paen [109] is illustrated in Appendix D (Fig. D.1). We discuss the concepts of UML-RT capsule diagrams and state machine diagrams (which, together, comprise the source metamodel) using the examples demonstrated by Posse and Dingel [120] and with reference to the corresponding classes in the UML-RT metamodel (Fig. D.1). We do so, by stating the UML-RT concept followed by the name of the
corresponding class from Fig. D.1 italicized and in brackets, e.g., a protocol (class \textit{Protocol}). More details on UML-RT capsule diagrams and state machine diagrams have been discussed in depth in the literature [133, 135, 134, 120].

\textbf{Structure Diagrams or Capsule Diagrams} In UML-RT, the structure of a system is specified as a UML-RT structure diagram or a capsule diagram. Fig. 6.4 shows an example of a UML-RT capsule diagram [120].

![Figure 6.4: An exemplar UML-RT capsule diagram [120].](image)

A UML-RT capsule diagram contains one or more (possibly hierarchical) capsules (class \textit{Capsule}) or components. Capsules communicate with each other by exchanging signals (class \textit{Signal}) via ports (class \textit{Port}), e.g., ports \textit{p1} and \textit{p2} in Fig. 6.4. Ports can be linked by connectors (class \textit{PortConnector}) (e.g., connectors \textit{l1} and \textit{l2} in Fig. 6.4). Communicating ports can also be initially unwired and they can be linked dynamically at run-time. Each port implements a protocol (class \textit{Protocol}), which identifies outgoing and incoming signals (classes \textit{SignalType}, \textit{OUT1}, \textit{IN0}) that can be sent and received via the port. A pair of connected ports must implement the same
protocol and the output signals of one port must be the input signals of the other port, and vice versa. In this case, one of the ports is said to be the base port and the other port is the conjugate port (classes PortType, BASE0, and CONJUGATE1). In Fig. 6.4 for example, ports p6 and p9 must implement the same protocol to be able to exchange signals with each other. Port p6 is the base port and port p9 is the conjugate port (annotated with ~).

Ports can be external end ports, external relay ports, or internal ports. External end ports are ports linked to external capsules and used by the capsule’s state machine to send or receive messages (e.g., port p2 in Fig. 6.4). External relay ports are ports that connect a capsule to some sub-capsule (e.g., ports p1 and p3 in Fig. 6.4). Internal or protected ports are used to communicate between the capsules state machine and some sub-capsule (e.g., port p4 in Fig. 6.4).

A capsule may contain sub-capsules, where each sub-capsole may have a fixed role, an optional role, or a plug-in role (classes CapsuleRole, RoleType, FIXED0, OPTIONAL1, and PLUGIN2). A fixed sub-capsule is created (or destroyed) when its containing capsule is created (or destroyed). An optional sub-capsule may be created or destroyed at a different time than that of its owning capsule. Plug-in capsule roles are placeholders for capsules, which can be filled and removed dynamically. In Fig. 6.4, B is a fixed capsule; C is an optional role; and D is a plug-in capsule role, where the different roles are differentiated by different colors.

Each capsule is assigned to a logical thread (classes Thread and LogicalThread) which is a concurrent thread of execution. Each logical thread is assigned to some physical thread (class PhysicalThread) which is the actual processing thread used by the underlying platform. Several capsules (or logical threads) can share the same
6.2. THE UMLRT-TO-KILTERA CASE STUDY

Figure 6.5: An exemplar UML-RT state machine diagram [120].

physical thread. Each physical thread has a controller that manages the execution of all capsules (or logical threads) within the physical thread. A controller contains the event pool of all events whose intended receiver is a capsule associated with the physical thread. A controller also enforces run-to-completion semantics, i.e., each state machine of each capsule that the controller controls processes each event fully before processing the next event.

Behavioral State Machine Diagrams In UML-RT, a capsule’s behavior is specified as a state machine diagram. Fig. 6.5 shows an example of a UML-RT state machine diagram [120].

A UML-RT state machine diagram (class StateMachine) has one or more (possibly hierarchical) states (class State), e.g., state Active and state Locked in Fig. 6.5. States are traversed through transitions (class Transition), e.g., transition start and transition standby in Fig. 6.5. Transitions can be sibling transitions between states in the
same hierarchical level, incoming transitions from a state to one of its sub-states, or outgoing transitions from a sub-state to its containing state (classes TransitionType, SIBLING0, IN1, and OUT2).

Transitions can have triggers (class Trigger), where each trigger is composed of a signal (class Signal) received on a port (class Port). Transitions can have guards (i.e., boolean expressions) and guarded transitions are only triggered when the guards evaluate to true. A state can have entry and/or exit actions (class Action), i.e., actions that are executed when the state is entered or exited. A transition can have actions that are executed when the transition is triggered (class Action), i.e., actions that are executed after leaving the transition’s source state and before arriving at the transition’s target state such as action a1 on transition start in Fig. 6.5.

To represent a transition crossing boundaries of states, the transition must be broken up into segments where each segment links connection points, i.e., entry points or exit points (classes Vertex, EntryPoint, and ExitPoint). Connected segments form a transition chain, which is executed as one step at run-time due to UML-RT’s run-to-completion semantics, i.e., a state machine handles one event at a time and if a chain of transitions is triggered, the actions on the chain of transitions are fully executed before the next event is handled.

In UML-RT, all entry points are by default connected to deep-history states. For example, suppose that an entry point of a composite state n is the target of a transition and that the entry point is not linked to a sub-state of n. If state n has not been previously visited and there is an initial transition from state n’s initial point (class InitialPoint), then the initial transition is followed and the default target state is entered. If state n has been previously visited, then the last visited sub-state of
n is entered. If state n has not been previously visited and has no initial transition, then the state machine remains at the border of state n.

As part of a state’s entry or exit actions or a transition’s action, a capsule can schedule an event to occur after a specific amount of time if it has a port that implements a special timing protocol. After the specified time elapses, the capsule containing the port will receive a time-out signal from the port, after which the capsule executes the scheduled event.

**The Target Kiltera Metamodel**

Kiltera [116, 117, 118, 119, 121] is a language based on process calculus\(^2\) for modelling concurrent, interacting, real-time processes with support for mobility and distributed systems. The main notions in Kiltera are processes and channels. Processes can run concurrently and interact by asynchronous message passing over channels. Channels can also be sent as parts of messages. Events and channels are closely related concepts in Kiltera. For example, triggering an event named \(x\) is equivalent to sending a message over channel named \(x\), and listening to an event named \(y\) is equivalent to waiting for input over a channel named \(y\).

The complete Kiltera metamodel considered by Posse and Dingel [120] and manipulated by Paen [109] is illustrated in Appendix D (Fig. D.2). We summarize the syntax and semantics of Kiltera [120], and we discuss the mapping between the Kiltera concepts and their corresponding classes in the metamodel shown in Appendix D (Fig. D.2). A complete formal account of Kiltera was previously presented by Posse [117].

\(^2\)Process calculi are mathematical formalisms used to model concurrent systems with a broad range of analysis techniques.
Fig. 6.6 shows the syntax of Kiltera [120] where \( P \) is the set of Kiltera processes, \( G \) is the set of guards, \( D \) is the set of definitions, \( E \) is the set of expressions, and \( R \) is the set of patterns.

**Syntax**  
Fig. 6.6 shows the syntax of Kiltera [120] where \( P \) is the set of Kiltera processes, \( G \) is the set of guards, \( D \) is the set of definitions, \( E \) is the set of expressions, and \( R \) is the set of patterns.

**Semantics**  
We summarize the semantics of Kiltera as presented by Posse and Dingel [120]. Kiltera expressions \( E \) and patterns \( R \) can be constants (e.g., \texttt{true} and \texttt{false}), variables (e.g., \( x \)), and tuples of the form \( \langle E_1, \ldots, E_m \rangle \). Kiltera expressions \( E \) can additionally include function calls of the form \( f(E_1, \ldots, E_m) \).

Several Kiltera processes \( P \) are enumerated in Fig. 6.6. The process \texttt{stop} is a process that executes no action. The process \texttt{done} represents a successfully terminated
process. A trigger or an output process of the form \( a!E \) (where \( E \) is optional) outputs the value of expression \( E \) (or null, if \( E \) is not specified) over channel \( a \). Listener processes take the form \( \text{when}\{G_1 \rightarrow P_1| \ldots |G_n \rightarrow P_n\} \). In listener processes, \( G_i \) is an input guard which takes the form \( a_i?R_i@y_i \), where \( a_i \) is an event or channel, \( R_i \) is an optional pattern, and \( y_i \) is an optional variable. A listener process (of the form \( \text{when}\{G_1 \rightarrow P_1| \ldots |G_n \rightarrow P_n\} \)) listens to all channels (i.e., \( a_1 \ldots a_n \)) of the guards \( G_1 \ldots G_n \). When a specific channel \( a_i \) is triggered with a value that matches the corresponding pattern \( R_i \) of guard \( G_i \), three steps are carried out: (1) the corresponding process \( P_i \) is executed, (2) variable \( y_i \) of the corresponding guard \( G_i \) stores the amount of time waited by the listener, and (3) the alternative guards (and their corresponding processes) are ignored. The new process (of the form \( \text{new} a_1,\ldots,a_n \text{ in } P \)) creates the channels \( a_1 \ldots a_n \) that are private to process \( P \). Conditional processes in Kiltera have the standard semantics of conditionals. The delay process (of the form \( \text{wait} E \rightarrow P \)) waits for an amount of time equal to the value of expression \( E \) before executing process \( P \). Local definition processes of the form \( \text{def}\{D_1; \ldots ;D_n\} \text{ in } P \) declares the definitions \( D_1 \ldots D_n \) and executes \( P \), where the scope of \( D_1 \ldots D_n \) is the entire term. The definitions \( D_1 \ldots D_n \) can be process definitions, function definitions, or local variable definitions (explained in the following text). Parallel composition and sequential composition processes, as their names suggest, represent the parallel and sequential composition of the two processes in the term. The assignment process (of the form \( x := E \)) assigns the value of expression \( E \) to the variable \( x \).

Kiltera definitions can be function definitions, variable definitions, or process definitions. A function definition of the form \( \text{func } f(x_1,\ldots,x_n) = E \) defines a function \( f \) with parameters (i.e., \( x_1 \ldots x_n \)) that are used in the body of the function
(i.e., expression $E$). A variable definition of the form $\text{var } x = E$ defines a variable named $x$ and assigns the value of expression $E$ to $x$. A process definition of the form $\text{proc } A(x_1, \ldots, x_n) = P$ defines a process $A$ with parameters (i.e., $x_1 \ldots x_n$) that are used in the body of the process (i.e., process $P$). Thus, the instantiation process of the form $A(E_1, \ldots, E_n)$ instantiates a process defined by $\text{Proc } A(x_1, \ldots, x_n) = P$ where the parameters (i.e., $x_1 \ldots x_n$) are substituted in $P$ by the values of the expressions $E_1 \ldots E_n$.

Correspondence Between the Kiltera Concepts and Metamodel  We summarize how the Kiltera concepts described in this section correspond to classes in the Kiltera metamodel shown in Appendix D (Fig. D.2). We focus only on the classes manipulated in the DSLTrans implementation of the UML-RT-to-Kiltera transformation (described in Section 6.2.3).

Kiltera expressions and patterns are represented by classes $\text{Expr}$ and $\text{Pattern}$. Input guards of the form $a?R@y$ are represented by the $\text{ListenBranch}$ class with attributes $\text{channel}$ and $\text{after}$ to represent $a$ and $y$ in $a?R@y$. Class $\text{ListenBranch}$ also has an association named $\text{match}$ to class $\text{Pattern}$ to represent the $R$ in $a?R@y$.

All Kiltera processes are represented by the super class $\text{Proc}$ from which the following classes inherit to represent specific processes.

- The process $\text{done}$ is represented by the class $\text{Null}$.

- A trigger process (of the form $a!E$) is represented by class $\text{Trigger}$, with attribute $\text{channel}$ to (represent $a$ in $a!E$) and an association named $\text{output}$ with class $\text{Expr}$ (to represent $E$ in $a!E$).

- A listener process (of the form $\text{when}\{G_1 \rightarrow P_1 | \ldots | G_n \rightarrow P_n\}$) is represented by
class \textit{Listen}, where each input guard (i.e., each $G_i$ in $\text{when}\{G_1 \rightarrow P_1 | \ldots | G_n \rightarrow P_n\}$) is represented by an association named \textit{branches} with class \textit{ListenBranch} and each alternative process (i.e., each $P_i$ in $\text{when}\{G_1 \rightarrow P_1 | \ldots | G_n \rightarrow P_n\}$) is represented by an association named $p$ that class \textit{ListenBranch} has with class \textit{Proc}.

- The new process (of the form \textit{new} $a_1, \ldots, a_n$ in $P$) is represented by class \textit{New} with an association named \textit{channelNames} with class \textit{Name} (to represent $a_i$ in \textit{new} $a_1, \ldots, a_n$ in $P$) and an association named $p$ with class \textit{Proc} (to represent $P$ in \textit{new} $a_1, \ldots, a_n$ in $P$).

- The conditional processes is represented by the class \textit{ConditionSet}, which has an association named \textit{branches} with class \textit{ConditionBranch} (to represent the “if/then” clause) and an association named \textit{alternative} with class \textit{Proc} (to represent the “else” clause). Class \textit{ConditionBranch} has an association named \textit{if} with class \textit{Expr} to represent the “if” condition and an association named \textit{then} with class \textit{Proc} to represent the “then” clause.

- The delay process (of the form \textit{wait} $E \rightarrow P$) is represented by the class \textit{Delay} which has an association named \textit{time} with class \textit{Expr} (to represent $E$ in \textit{wait} $E \rightarrow P$) and an association named $p$ with class \textit{Proc} (to represent $P$ in \textit{wait} $E \rightarrow P$).

- Local definitions (of the form \textit{def}\{\textit{D}_1; \ldots; \textit{D}_n\} in $P$) are represented by class \textit{LocalDef} which has an association named \textit{def} with class \textit{Def} (to represent $\textit{D}_i$ in \textit{def}\{\textit{D}_1; \ldots; \textit{D}_n\} in $P$) and an association named $p$ with class \textit{Proc} (to represent $P$ in \textit{def}\{\textit{D}_1; \ldots; \textit{D}_n\} in $P$).
Parallel composition and sequential composition processes are represented by classes *Par* and *Seq*, respectively, where each of the two classes has an association named *p* with class *Proc* to represent the composed processes.

All Kiltera definitions are represented by the super class *Def*. Classes *FuncDef* and *ProcDef* inherit from class *Def* and are used to represent function definitions and process definitions. Class *ProcDef* has an attribute *name* to represent $A$ in $\text{proc } A(x_1, \ldots, x_n) = P$, an association named *channelNames* with class *Name* to represent $x_i$ in $\text{proc } A(x_1, \ldots, x_n) = P$, and an association named *p* with class *Proc* to represent $P$ in $\text{proc } A(x_1, \ldots, x_n) = P$. The instantiation process of the form $A(E_1, \ldots, E_n)$ is represented by class *Inst*. Class *Inst* has an attribute *name* to represent $A$ in $A(E_1, \ldots, E_n)$ and an association named *channelNames* with class *Name* to represent $E_i$ in $A(E_1, \ldots, E_n)$.

### 6.2.2 The UML-RT-to-Kiltera Model Transformation Mapping Rules

In this section, we summarize the required mapping between the source UML-RT metamodel and the target Kiltera metamodel, which was described in detail by Posse and Dingel [120]. Since Paen [109] implemented the UML-RT-to-Kiltera transformation [120] in ATL (focusing only on transforming state machines less the history states and the enabled transition selection policy) and since we will reimplement this transformation in DSLTrans for this case study, we only summarize the mapping rules [120] that are relevant to our DSLTrans implementation of the transformation (described in Section 6.2.3).

The transformation between the source UML-RT state machines and the target Kiltera metamodel should fulfill the following mapping rules:
1. A state $n$ is encoded as a process definition named $S_n$ with three parameters: $enp$, $exit$ and $exack$. Parameter $enp$ holds the name of the entry point used to enter state $n$. Parameter $exit$ holds the channel that will be used by state $n$’s parent to request state $n$ to exit. Parameter $exack$ holds the channel used by state $n$ to acknowledge an exit request to state $n$’s parent. We use the naming convention for channels used by Posse and Dingel [120] and in Paen’s [109] implementation where a channel named $x$ in a process $S_n$ (encoding state $n$, whose parent state is state $m$ and sub-state is state $w$) is used for communication between $S_m$ and $S_n$, whereas a channel with a primed name $x’$ in process $S_n$ is used for communication between $S_n$ and $S_w$.

2. Sub-states (e.g., $n_1$ and $n_2$) of a composite state $n$ are encoded as nested process definitions (i.e., sub-processes) of $S_n$ using the local definition process (Fig. 6.6), i.e., the body of process $S_n$ will be $\text{Def}\{S_{n_1}; S_{n_2}\}$ in $P$ where $P$ is a Kiltera process that reflects the behaviour of state $n$.

3. Entering a state $n$ (represented as process $S_n$) is encoded by instantiating $S_n$ using the instantiation process (Fig. 6.6) of the form $S_n(E_1, \ldots, E_n)$ (where $E_i$ are the expressions passed as arguments to instantiate $S_n$).

4. To encode simple transitions with triggers (i.e., signals received on a capsule’s port) in a state $n$, process $S_n$ must contain a sub-process $\text{Handler}$ which receives any incoming signal and automatically decides on the action to take based on any incoming signal. Thus, sub-process $\text{Handler}$ operates as follows:

(a) For a non-composite state $n$ (encoded as process $S_n$) with a parent state $m$ (encoded as process $S_m$), the branches of the $\text{Handler}$ sub-process of $S_n$
perform the following steps: (i) If an exit request is received on the exit channel from $S_m$, then the $Handler$ executes state $n$’s exit action and sends an acknowledgement on the $exack$ channel; (ii) if a signal that triggers one of state $n$’s transitions is received, the $Handler$ sequentially executes the state $n$’s exit action, the transition’s action, and the process corresponding to the transition’s target (state or exit point).

(b) For a composite state $n$ (encoded as process $S_n$) with a parent state $m$ (encoded as process $S_m$), the branches of the $Handler$ sub-process of $S_n$ perform the following steps: (i) If an exit request is received on the exit channel from $S_m$, $S_n$ requests its currently active sub-state to exit using the $exit'$ channel and waits for an acknowledgement on the $exack'$ channel. After an acknowledgement is received on the $exack'$ channel, the $Handler$ executes state $n$’s exit action and sends an acknowledgement to $S_m$ on the $exack$ channel; (ii) if a stop handler request is received on the $sh'$ channel when leaving state $n$ through one of its exit points, a $done$ process is executed to indicate successful termination of $S_n$; (iii) if a signal that triggers one of state $n$’s transitions is received, the $Handler$ sends an exit request to state $n$’s currently active sub-state on the $exit'$ channel. When the sub-state sends an acknowledgement on the $exack'$ channel, the $Handler$ sequentially executes state $n$’s exit action, the transition’s action, and the process corresponding to the transition’s target (state or exit point).

5. A process $S_n$ that encodes a non-composite state $n$ with no outgoing transitions does not define a $Handler$. Instead, $S_n$ executes a $done$ process to indicate that state $n$ does not handle any incoming signals and that $S_n$ successfully terminates
once the corresponding state $n$ is entered.

6. When composite state $n$ is entered, the choice of the sub-state to enter next is encoded using a sub-process called *Dispatcher* of process $S_n$. If state $n$ is entered through an entry point (identified by the argument passed as parameter $enp$ to the *Dispatcher*) that is explicitly connected to a sub-state $v$, then the *Dispatcher* instantiates $S_v$ after executing the transition’s action. If, however, state $n$ is entered through an entry point that is not explicitly connected to a sub-state, then the *Dispatcher* follows state $n$’s initial transition (i.e., executes the initial transition’s action) and enters the initial sub-state $i$ (i.e., instantiates $S_i$). Thus, the *Dispatcher* uses the argument passed to its $enp$ parameter to identify the sub-state $v$ connected to the corresponding entry point (if any) and executes $S_v$.

7. For process $S_n$ to accurately reflect the behavior of a composite state $n$, the body of $S_n$ has to execute (i.e., instantiate) the *Handler* and the *Dispatcher* sub-processes in parallel. This ensures that state $n$ can constantly handle (in parallel) incoming signals and invoke the relevant sub-state when state $n$ is entered. Both the *Handler* and the *Dispatcher* have to be instantiated with primed channels (e.g., $exit'$, $exact'$, and $sh'$) passed as arguments, to allow both sub-processes to interact with the currently active sub-state of state $n$.

8. Exit points of non-composite states are not encoded at all. An exit point $x$ of a composite state $n$ is encoded as a sub-process $B_x$ of process $S_n$. Sub-process $B_x$ jumps to the final target state of the transition (whose target is the exit point) by executing two steps in parallel:
6.2. THE UMLRT-TO-KILTERA CASE STUDY

(a) $B_x$ stops the Handler of $S_n$ to deactivate the state containing exit point $x$. This is done by defining a channel $sh'$ (short for stop handler) for $B_x$ and sending a stop handler request on $sh'$.

(b) $B_x$ executes the transition’s action followed by the process corresponding to the final target (state or exit point) of the transition.

9. An outgoing transition of a composite state $n$ (represented as a process $S_n$) whose target is an exit point $x$ of state $n$ (represented as sub-process $B_x$ of $S_n$) is encoded as an instantiation of process $B_x$.

10. A group transition is a transition whose source is an exit point of a composite state. When a group transition is triggered, the currently active sub-state must become inactive. To encode the effect of triggering a group transition from a composite state $n$ (encoded as process $S_n$), the Handler sub-process of process $S_n$ sends an exit request on the $exit'$ channel (to ask its currently active sub-state to exit) and waits for an acknowledgement on the $exack'$ channel before it proceeds to the target of the group transition.

The original mapping from UML-RT to Kiltera [120] did not provide a translation of UML-RT state machine actions (i.e., state entry actions, state exit actions, and transition actions), nor was such a translation reflected in Paen’s [109] ATL implementation of the UML-RT-to-Kiltera transformation. This is mainly because the action language may vary. Hence, the studies [120, 109] assumed that an appropriate translation was provided based on the used action language. Posse and Dingel [120] only discussed where such a translation should be placed in the Kiltera output. For example, entry actions of a state $n$ (represented as a process $S_n$) should be encoded
at the beginning of $S_n$ to ensure that the actions are executed once state $n$ is entered (i.e., when $S_n$ is instantiated). Exit actions of a state $n$ should be encoded at the beginning of every sub-process $B_{x_i}$ of $S_n$ (corresponding to exit points $x_i$ of state $n$). Transition actions of state $n$ should be encoded in each branch of the $S_n$’s Handler sub-process, as described in mapping rule 4 shown above.

6.2.3 Reimplementation of the UMLRT-to-Kiltera Model Transformation in DSLTrans

To achieve the mapping between UML-RT state machines and Kiltera process models described by Posse and Dingel [120] and summarized in Section 6.2.2 above, Paen [109] implemented the UMLT-RT-to-Kiltera transformation in ATL (focusing only on transforming state machines less the history states and the enabled transition selection policy [120]). We reimplement the UML-RT-to-Kiltera model transformation in DSLTrans to facilitate verifying properties of the transformation using our symbolic model transformation property prover. The complete DSLTrans implementation of the UML-RT-to-Kiltera transformation is shown in Fig. 6.7. Table 6.3 summarizes the rules in each layer of the transformation shown in Fig. 6.7, the input and output types that are mapped and created by each rule, and the mapping rules (from Section 6.2.2) that the transformation rules implement. In what follows, we discuss the implementation shown in Fig. 6.7 and summarized in Table 6.3 in detail.

In ‘layer 1’ of Fig. 6.7, rule ‘State2ProcDef’ implements rule 1 in Section 6.2.2 by mapping any state into a process definition (i.e., $ProcDef$ element) with the following parameters (i.e., $Name$ elements with the following $literal$ attributes): $enp$, $exit$, and $exack$. Rule ‘State2ProcDef’ follows the naming convention stated in rule 1 in
Figure 6.7: Layers 1-3 of the initial UML-RT-to-Kiltera transformation implementation (figure continued in the next page).
Figure 6.7: Layers 4-6 of the initial UML-RT-to-Kiltera transformation implementation.
Table 6.3: The rules in each layer of the UML-RT-to-Kiltera transformation after reimplementing the transformation in DSLTrans, their input and output types, their corresponding rules from Section 6.2.2.

Section 6.2.2, i.e., a state named \( n \) is mapped into a process named \( S_n \).

In ‘layer 2’ of Fig. 6.7, rule ‘MapBasicStateNoTrans’ implements rule 5 in Section 6.2.2 by adding to the body of any process definition (i.e., \( \text{ProcDef} \) element) that was previously generated from a non-composite state that does not have outgoing transitions, a \textit{done} sub-process (i.e., \( \text{Null} \) element). The fact that rule ‘MapBasicStateNoTrans’ only maps \( \text{ProcDef} \) elements that were previously generated from input \( \text{State} \) elements is shown by the backward link (appearing as a vertical, dashed
6.2. THE UMLRT-TO-KILTERA CASE STUDY

line and described in Section 5.1) between the ProcDef element and the State element in rule ‘MapBasicStateNoTrans’ (Fig. 6.7). Rule ‘MapBasicState’ (together with rule ‘Trans2ListenBranch’ in ‘layer 5’ of Fig. 6.7) implements rule 4a from Section 6.2.2 by handling signals (e.g., exit requests from the containing state or transition triggers) for non-composite states with outgoing transitions. Rule ‘MapBasicState’ adds to the body of any process definition (i.e., ProcDef element) that was previously generated (as shown by the backward link) from a non-composite state with outgoing transitions, a listener process (i.e., Listen element) with an input guard (i.e., ListenBranch element) that triggers an acknowledgement on the exack channel when it receives an exit request on the exit channel (Rule 10 in Section 6.2.2). Thus, the Listen element created by rule ‘MapBasicState’ represents the Handler sub-process.

Rule ‘MapCompositeState’ implements rule 7 in Section 6.2.2 by adding two elements to any ProcDef element \( S_n \) that was previously generated from a composite state \( n \): (i) a parameter \( sh \) (i.e., Name element with a literal named \( sh \)) to be used when exiting through an exit point of the state (Rule 8a in Section 6.2.2); and (ii) a local definition process (i.e., LocalDef element) to represent the body of \( S_n \). The body of the local definition process is a new process that creates channels exit\_in, exack\_in, and \( sh\_in \) (i.e., corresponding to exit’, exack’, and \( sh’ \))\(^3\) that are private to a parallel process. The parallel process executes or instantiates the Handler sub-process (i.e., Inst element named Handler) and the Dispatcher sub-processes (i.e., Inst element named Dispatcher) in parallel, while passing the primed channels as parameters to the two sub-processes.

In ‘layer 3’ of Fig. 6.7, rule ‘ExitPoint2ProcDef’ implements rule 8a in Section 6.2.2

\(^3\)We use a naming convention where a channel named, for example, \( sh’ \) in Section 6.2.2 is named as \( sh\_in \) in Fig. 6.7.
by adding to any local definition (i.e., \textit{LocalDef} element) that was previously generated from a composite state with an exit point \( v \), a process definition (i.e., \textit{ProcDef} element) named \( B_v \). Process \( B_v \) defines a parameter \( sh\_in \) and executes two processes in parallel (i.e., \textit{Par} element), one of which is triggering channel \( sh\_in \). The second process to be run in parallel is executing or instantiating the process corresponding to the final destination of the transition (implemented by the ‘MapExitWithTrans’ rule in ‘layer 5’ of Fig. 6.7 and reflects mapping rule 8b in Section 6.2.2). Rule ‘State2Handler’ implements rules 4b from Section 6.2.2 by adding to any local definition (i.e., \textit{LocalDef} element) that was previously generated from a composite state, a \textit{Handler} sub-process definition (i.e., \textit{ProcDef} element named \textit{Handler}). \textit{ProcDef Handler} has three parameters (i.e., \textit{Name} elements with the following \textit{literal} attributes): \( exit\_in \), \( exack\_in \), and \( sh\_in \). The body of \textit{ProcDef Handler} is a listener process (i.e., \textit{Listen} element) with branches or input guards (i.e., \textit{ListenBranch} elements) to handle the following incoming events:

- An incoming event on channel \( sh\_in \) executes a \textit{done} process (i.e., a \textit{Null} element).

- An incoming event on the \textit{exit} channel results in the state sequentially (i.e., \textit{Seq} element) triggering an event on the \textit{exit\_in} channel (to request its currently active sub-state to exit) and then waiting for an acknowledgement on the \textit{exack\_in} channel. If a response is received on the \textit{exack\_in} channel, the state triggers an output on the \textit{exack} channel to confirm successful exit of the state (and its previously active sub-state).

Additional branches or input guards to handle transition triggers are implemented by rule ‘Trans2HLListenBranch’ in ‘layer 5’. Rule ‘State2Dispatcher’ implements rule 6
from Section 6.2.2 by adding to any local definition (i.e., \textit{LocalDef} element) that was previously generated from a composite state with an initial transition, a \textit{Dispatcher} sub-process definition (i.e., \textit{ProcDef} element named \textit{Dispatcher}). \textit{ProcDef Dispatcher} has four parameters (i.e., \textit{Name} elements with the following \textit{literal} attributes): \textit{exit}, \textit{exack}, \textit{enp}, and \textit{sh}. The body of \textit{ProcDef Dispatcher} is a conditional process (i.e., \textit{ConditionSet} element) that instantiates (i.e., \textit{Inst} element) the process definition corresponding to the initial transition’s target state, if the state is entered through an entry point that is not explicitly connected to a sub-state. However, if the state is entered through an entry point that is explicitly connected to a sub-state \(v\), process \(S_v\) is instantiated (implemented by rule ‘MapStatesINtrans’ in ‘layer 5’ of Fig. 6.7).

In ‘layer 4’ of Fig. 6.7, rule ‘Trans2InstSIB’ implements rule 3 from Section 6.2.2 (for sibling transitions) by mapping any sibling transition whose destination is an entry point \(e\) of a state \(n\) to an instantiation process (i.e., \textit{Inst} element) named \(S_n\) with three channels (i.e., \textit{exit}, \textit{exack}, and \textit{sh}) and the name of the entry point (i.e., \(e\)) passed as parameters. Rule ‘Trans2InstOUT’ implements rule 9 from Section 6.2.2 by mapping any outgoing transition whose destination is an exit point \(x\) of state \(n\) to an instantiation process (i.e., \textit{Inst} element) named \(B_x\) with channel \textit{sh} (i.e., \textit{Name} element with a \textit{literal} attribute value of \textit{sh}) passed as a parameter. Rule ‘Trans2Inst’ implements rule 3 from Section 6.2.2 (for incoming transitions) by mapping any incoming transition of a composite state \(n\) whose destination is an entry point \(e\) of some sub-state \(b\) to an instantiation process (i.e., \textit{Inst} element) named \(S_b\) with three channels (i.e., \textit{exit\_in}, \textit{exack\_in}, and \textit{sh\_in}) and the name of the entry point (i.e., \(e\)) passed as parameters.
In ‘layer 5’ of Fig. 6.7, rule ‘Trans2ListenBranch’ implements rule 4a in Section 6.2.2 by handling transition triggers. In other words, for a non-composite state that has been previously mapped to a listener process (i.e., \textit{Listen} element), rule ‘Trans2ListenBranch’ maps every outgoing transition (previously mapped to an instantiation of process $S_w$) triggered by signal $x$ into a branch or an input guard (i.e., \textit{ListenBranch} element) of the listener process that waits for input on channel $x$ to execute $S_w$. Rule ‘MapExitWithTrans’ in Fig. 6.7 implements rule 8b in Section 6.2.2 by executing the process corresponding to the final target state of an outgoing transition whose destination is an exit point. The rule does so by connecting any parallel process previously generated from an \textit{ExitPoint} element to an instantiation process (i.e., \textit{Inst} element) previously generated from an outgoing transition of the exit point.

Rule ‘Trans2HListenBranch’ implements rules 4b and 10 from Section 6.2.2 by adding to any previously generated listener process (i.e., \textit{Listen} element) corresponding to a composite state, a branch or an input guard (i.e., \textit{ListenBranch} element) to handle events that trigger a (group) transition of the state. Two steps are carried out when a (group) transition (previously mapped to an instantiation of process $S_w$) in a composite state (previously mapped to a \textit{Listen} element) is triggered by signal $x$:

- The new input guard (i.e., \textit{ListenBranch} element) sequentially (i.e., \textit{Seq} element) sends an exit request to its active sub-state on channel $exit_{in}$ and then waits for the sub-state to send an acknowledgement on channel $exack_{in}$.

- Once an acknowledgement is received on channel $exack_{in}$, the process $S_w$ is executed or instantiated.
Rule ‘MapStatesINtrans’ implements rule 6 from Section 6.2.2 by adding to any conditional process (i.e., *ConditionSet* element) previously generated from a composite state *n*, a *ConditionBranch* element for every incoming transition (previously mapped to an instantiation of, for example, process *S_w*) whose source is an entry point *e* of state *n*. The *ConditionBranch* executes or instantiates *S_w* if entry point *e* is used to enter state *n* (as shown by the literal attribute of the *Expr* element; ‘enp=’+*Vertex*.name).

In ‘layer 6’ of Fig. 6.7, rule ‘MapNesting’ implements rule 2 in Section 6.2.2 by nesting two previously generated processes that correspond to two nested states.

Similar to the implementation of the GM-to-AUTOSAR model transformation in DSLTrans (described in Section 6.1.1), we use a combination of *Any* and *Exists* match elements (explained in Section 5.1) to represent positive application conditions (PACs) in our UML-RT-to-Kiltera model transformation rules. For example, rule ‘State2Dispatcher’ in layer 3 of Fig. 6.7 maps every composite *State* element to twelve elements, only if the composite *State* element has at least one initial *Transition* whose destination is the *EntryPoint* of a *StateMachine* element (or an element of a subclass of *StateMachine*). Thus, the MatchModel of rule ‘State2Dispatcher’ has a *State* element (represented as an *Any* match element) connected to a *Transition* element, an *EntryPoint* element, and a *StateMachine* element (represented as *Exists* match elements). Similarly, rule ‘MapExitWithTrans’ in layer 5 of Fig. 6.7 maps every *ExitPoint* element (represented as *Any* match element) that has been previously mapped to a *Par* element (as shown by the backward link and the free variable *parexitpoint*), only if the *ExitPoint* has at least one outgoing *Transition* (represented as an *Exists* match element) that has been previously mapped to an *Inst* element (as
shown by the backward link and the free variable \textit{instfortrans}).

As previously described in Section 5.1, we use a combination of free variables and backward links in the UML-RT-to-Kiltera transformation (Fig. 6.7) to allow several transformation rules to refer to the same generated element. For example, the ‘State2ProcDef’ rule in layer 1 (Fig. 6.7) binds the generated \textit{ProcDef} element to the free variable \textit{procdef}. Any successive rule such as the ‘MapBasicStateNoTrans’ rule in layer 2 can have a \textit{ProcDef} element with a free variable \textit{procdef} in their ApplyModel and connected with a backward link to a \textit{State} element in the rule’s MatchModel. This usage of the free variable \textit{procdef} and the backward link will make the ‘MapBasicStateNoTrans’ rule in layer 2 only match \textit{ProcDef} elements that have been previously generated from \textit{State} elements by the ‘State2ProcDef’ rule in layer 1.

6.2.4 Properties of Interest for the UMLRT-to-Kiltera Model Transformation

The properties of interest that we formulated for the UML-RT-to-Kiltera transformation are summarized in Table 6.4. We divide the formulated properties into four categories: \textit{Multiplicity Invariants}, \textit{Syntactic Invariants}, \textit{Pattern Contracts}, and \textit{Rule Reachability}. We use the same definitions for \textit{invariants} and \textit{contracts} as those used in Section 4.2.3, i.e., an invariant is a property defined on elements of the target metamodel only and a contract is a property that relates elements of the source and target metamodels. We add to the beginning of each property in Table 6.4 an abbreviation (e.g., \textit{(MM1)}, \textit{(SS2)}) that we will use to refer to the corresponding property.

Multiplicity Invariants ensure that the transformation does not produce an output
### Properties of interest for the UML-RT-to-Kiltera transformation

#### Multiplicity Invariants:
- (MM1) Every `New` is associated to `1..1 Proc` through the `p` association.
- (MM2) Every `LocalDef` is associated to `1..1 Proc` through the `p` association.
- (MM3) Every `ConditionBranch` is associated to `1..1 Expr` through the `if` association.
- (MM4) Every `ConditionBranch` is associated to `1..1 Proc` through the `then` association.
- (MM5) Every `Listen` is associated to `1..* ListenBranch` through the `branches` association.
- (MM6) Every `New` is associated to `1..* Name` through the `channelNames` association.
- (MM7) Every `LocalDef` is associated to `1..* Def` through the `def` association.
- (MM8) Every `ListenBranch` is associated to `0..1 Pattern` through the `match` association.
- (MM9) Every `Trigger` is associated to `0..1 Expr` through the `output` association.
- (MM10) Every `ConditionSet` is associated to `0..1 Proc` through the `alternative` association.
- (MM11) Every `Par` is associated to `2..* Proc` through the `p` association.

#### Syntactic Invariants:
- (SS1) If an `Inst` named `Handler` is created, then a `ProcDef` named `Handler` is also created.
- (SS2) If an `Inst` named `Dispatcher` is created, then a `ProcDef` named `Dispatcher` is also created.
- (SS3) If an `Inst` named `Handler` is associated to three `Name` elements, then a `ProcDef` named `Handler` is also created and is associated to three `Name` elements.

#### Pattern Contracts:
- (PP1) Every sibling `Transition` that has a `Trigger` with a `Signal` (in the input) is mapped to a `ListenBranch` that waits for input on a `channel` to execute the `(Inst)` (corresponding to the input `Transition`) with four `Name` elements passed as `channelNames` (in the output).
- (PP2) Every outgoing `Transition` that has a `Trigger` with a `Signal` (in the input) is mapped to a `ListenBranch` that waits for input on a `channel` to execute the `(Inst)` (corresponding to the input `Transition`) with one `Name` element passed as a `channelName` (in the output).
- (PP3) Every `ExitPoint` of a composite `State` with a sibling `Transition` (in the input) is mapped to a `Par` where the two parallel processes are the `Inst` (corresponding to the input `Transition`) and a `Trigger` which produces output on `channel sh_m` (in the output).
- (PP4) Every `ExitPoint` of a composite `State` with an outgoing `Transition` (in the input) is mapped to a `Par` where the two parallel processes are the `Inst` (corresponding to the input `Transition`) and a `Trigger` which produces output on `channel sh_m` (in the output).
- (PP5) Nested `State` elements are mapped to nested `ProcDef` elements.

### Rule Reachability

<table>
<thead>
<tr>
<th>Rule</th>
<th>Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RR1)</td>
<td><code>Reachable(State2ProcDef)</code></td>
</tr>
<tr>
<td>(RR2)</td>
<td><code>Reachable(NapBasicStateNoTrans)</code></td>
</tr>
<tr>
<td>(RR3)</td>
<td><code>Reachable(NapBasicState)</code></td>
</tr>
<tr>
<td>(RR4)</td>
<td><code>Reachable(NapCompositeState)</code></td>
</tr>
<tr>
<td>(RR5)</td>
<td><code>Reachable(ExitPoint2ProcDef)</code></td>
</tr>
<tr>
<td>(RR6)</td>
<td><code>Reachable(State2Handler)</code></td>
</tr>
<tr>
<td>(RR7)</td>
<td><code>Reachable(State2Dispatcher)</code></td>
</tr>
<tr>
<td>(RR8)</td>
<td><code>Reachable(Trans2InstrIB)</code></td>
</tr>
<tr>
<td>(RR9)</td>
<td><code>Reachable(Trans2InstrOUT)</code></td>
</tr>
<tr>
<td>(RR10)</td>
<td><code>Reachable(Trans2Instr)</code></td>
</tr>
<tr>
<td>(RR11)</td>
<td><code>Reachable(Trans2ListenBranch)</code></td>
</tr>
<tr>
<td>(RR12)</td>
<td><code>Reachable(NapExitWithTrans)</code></td>
</tr>
<tr>
<td>(RR13)</td>
<td><code>Reachable(Trans2HListenBranch)</code></td>
</tr>
<tr>
<td>(RR14)</td>
<td><code>Reachable(NapStatesINtrans)</code></td>
</tr>
<tr>
<td>(RR15)</td>
<td><code>Reachable(NapNesting)</code></td>
</tr>
</tbody>
</table>

Table 6.4: Properties of interest for the UML-RT-to-Kiltera transformation.
that violates the multiplicities in the Kiltera metamodel (Fig. D.2). We verify all the multiplicity invariants of the Kiltera metamodel that are shown in Fig. D.2 as annotations on associations. We do not check ‘0..*’ multiplicity invariants since they specify no constraints on the number of associated elements and hence, such invariants will always hold. We identify eleven multiplicity invariants that are not ‘0..*’ multiplicity invariants. 

\[ \text{MM1} \] ensures that each \textit{New} element is associated to one \textit{Proc} element through the \textit{p} association. 
\[ \text{MM2} \] ensures that each \textit{LocalDef} element is associated to one \textit{Proc} element through the \textit{p} association. 
\[ \text{MM3} \] ensures that each \textit{ConditionBranch} element is associated to one \textit{Expr} element through the \textit{if} association. 
\[ \text{MM4} \] ensures that each \textit{ConditionBranch} element is associated to one \textit{Proc} element through the \textit{then} association. 
\[ \text{MM5} \] ensures that each \textit{Listen} element is associated to at least one \textit{ListenBranch} element through the \textit{branches} association. 
\[ \text{MM6} \] ensures that each \textit{New} element is associated to at least one \textit{Name} element through the \textit{channelNames} association. 
\[ \text{MM7} \] ensures that each \textit{LocalDef} element is associated to at least one \textit{Def} element through the \textit{def} association. 
\[ \text{MM8} \] ensures that each \textit{ListenBranch} element is associated to maximum one \textit{Pattern} element through the \textit{match} association. 
\[ \text{MM9} \] ensures that each \textit{Trigger} element is associated to maximum one \textit{Expr} element through the \textit{output} association. 
\[ \text{MM10} \] ensures that each \textit{ConditionSet} element is associated to maximum one \textit{Proc} element through the \textit{alternative} association. 
\[ \text{MM11} \] ensures that each \textit{Par} element is associated to two or more \textit{Proc} elements through the \textit{p} association.

Syntactic Invariants ensure that the generated Kiltera output model is well-formed with respect to Kiltera’s syntax. Both \textit{SS1} and \textit{SS2} ensure that if an \textit{Inst} element with a specific name (i.e., \textit{Handler} or \textit{Dispatcher}) is created, then a \textit{ProcDef} element
with the same name (i.e., Handler or Dispatcher) is also created. In other words, SS1 and SS2 ensure that for a process named Handler or Dispatcher to be instantiated (shown as an Inst object), then a process with the same name must be defined (shown as a ProcDef object). SS3 ensures that if an Inst element named Handler is associated with three Name elements, then a ProcDef element named Handler is also created and is associated to three Name elements. In other words, SS3 ensures that if a process is instantiated by passing to it three arguments or channel names, then a process definition with the same name exists and takes three parameters.

Pattern contracts require that if a certain pattern of elements exists in the input model, then a corresponding pattern of elements exists in the output model. PP1 and PP2 mandate that if a sibling or an outgoing Transition has a Trigger with a Signal (in the input model), then a ListenBranch that waits for input on some channel to execute the Inst corresponding to the input Transition will eventually be created in the output model. PP3 and PP4 mandate that if an ExitPoint of a composite State has a sibling or an outgoing Transition (in the input model), then a parallel process (class Par) will be created in the output model to execute two processes in parallel: (i) a Trigger which outputs an empty message on channel sh_in, and (ii) the Inst corresponding to the matched Transition. PP5 ensures that a hierarchy of States is mapped to a hierarchy of ProcDefs. Pattern contracts can be thought of as a means to check that if an input model has the match pattern of more than one rule, then all the applicable rules will execute and produce the expected output. For example, PP3 checks that if the match pattern of three rules (i.e., ‘ExitPoint2ProcDef’, ‘Trans2InstSIB’, and ‘MapExitWithTrans’) exists in a path condition, then the corresponding, combined output of these three rules is in the same path condition. Pattern
contracts can also be thought of as a means to check that the transformation layer achieved its purpose by formulating the layer’s purpose as a property to be verified. For example, the purpose of ‘layer 6’ is to create the hierarchy between the nested ProcDefs that correspond to nested States. PP5 checks for that property and hence, allows us to verify that ‘layer 6’ achieved its intended purpose.

Rule reachability is a property that checks whether a specific rule exists in any of the generated path conditions. In other words, a rule is said to be reachable if it is triggered in any execution path or path condition of the transformation being verified. A rule that is not reachable is said to be a dead rule and indicates a transformation bug that needs to be fixed. Since the UML-RT-to-Kiltera transformation has 15 rules (as shown in Fig. 6.7), we formulate 15 reachability checks, i.e., \((RR_1)\ldots(RR_{15})\), as shown in Table 6.4.

6.2.5 Formulation of Properties for the UMLRT-to-Kiltera Model Transformation

In Section 6.2.4, we described properties of interest for the UML-RT-to-Kiltera transformation and we summarized them in Table 6.4. In this section, we demonstrate how we formulate the properties shown in Table 6.4 in our property prover. Since properties that belong to the same property type in Table 6.4 are formulated similarly, we demonstrate in this section the formulation of one property from each property type. The formulation of all the properties in Table 6.4 is demonstrated in Appendix C.

We demonstrate the formulation of ‘1..1’ multiplicity invariants (e.g., \(MM_1\), \(MM_2\), \(MM_3\), and \(MM_4\) in Table 6.4) by showing the formulation of \(MM_1\) in Fig. 6.8 as an example. \(MM_1\) ensures that if a New element is created in the output model,
6.2. THE UMLRT-TO-KILTERA CASE STUDY

Figure 6.8: AtomicContracts AC1, AC2, and AC3 that are used to express MM1 (Table 6.4) as $AC1 \implies_{tc} (AC2 \land_{tc} \neg_{tc} AC3)$.

Figure 6.9: AtomicContracts AC4 and AC5 that are used to express MM5 (Table 6.4) as $AC4 \implies_{tc} AC5$.

then this specific New element must be connected to one Proc element (and not more). Using the AtomicContracts in Fig. 6.8, MM1 can be expressed as $AC1 \implies_{tc} (AC2 \land_{tc} \neg_{tc} AC3)$. The free variable 'NEW' in Fig. 6.8 mandates that if AC1 holds for a specific New element, then AC2 should hold and AC3 should not hold for the same New element. For ‘1..*’ multiplicity invariants (e.g., MM5, MM6, and MM7 in Table 6.4), we demonstrate the formulation of MM5 in Fig. 6.9 as an example. MM5 ensures that if a Listen element is created in the output model, then this specific Listen element must be connected to at least one ListenBranch element. Using the AtomicContracts in Fig. 6.9, MM5 can be expressed as $AC4 \implies_{tc} AC5$, where the free variable 'LISTEN' in Fig. 6.9 is used to refer to the same matched Listen.
element. For ‘0..1’ multiplicity invariants (e.g., MM8, MM9, and MM10 in Table 6.4), we demonstrate the formulation of MM8 in Fig. 6.10 as an example. MM8 ensures that if a ListenBranch element is created in the output model, then this specific ListenBranch element must be connected to either one or no Pattern elements. Using the AtomicContracts in Fig. 6.10, MM8 can be expressed as AC6 \implies tc ((AC7 \land tc \neg tc AC8) \lor tc (\neg tc AC7)), where the free variable ‘LISTENBR’ in Fig. 6.10 is used to refer to the same matched ListenBranch element. For the formulation of ‘2..*’ multiplicity invariants (e.g., MM11 in Table 6.4), we demonstrate the formulation of MM11 in Fig. 6.11. MM11 ensures that if a Par element is created in the output model, then this specific Par element must be connected to at least two Proc elements. Using the AtomicContracts in Fig. 6.11, MM11 can be expressed as AC9 \implies tc AC10, where the free variable ‘PAR’ in Fig. 6.11 is used to refer to the same matched Par element.

Using the AtomicContracts in Fig. 6.12, the syntactic invariant SS1 can be expressed as AC11 \implies tc AC12. The former propositional formula can be interpreted as ‘If the output has an Inst element named Handler (AC11), then the same output must have the same Inst element accompanied with a ProcDef element named
6.2. THE UMLRT-TO-KILTERA CASE STUDY

Figure 6.11: AtomicContracts AC9 and AC10 that are used to express MM11 (Table 6.4) as $AC9 \rightarrow_{tc} AC10$.

Figure 6.12: AtomicContracts AC11 and AC12 that are used to express SS1 (Table 6.4) as $AC11 \rightarrow_{tc} AC12$.

Handler (AC12)”. The free variable ‘INST’ in Fig. 6.12 mandates that if AC11 holds for a specific Inst element, then AC12 should also hold for the same Inst element. The syntactic invariant SS3 can be expressed as $(AC13 \land_{tc} \neg_{tc} AC14) \rightarrow_{tc} (AC15 \land_{tc} \neg_{tc} AC16)$ where AC13, AC14, AC15, and AC16 are the AtomicContracts shown in Fig. 6.13. The former propositional formula can be interpreted as “If the output has an Inst element named Handler that is connected to three Name elements (AC13) and not more than three (AC14), then the same output must have the same Inst element accompanied with a ProcDef element named Handler that is connected to three Name elements (AC15) and not more than three (AC16)”. The free variable ‘INST’ in Fig. 6.13 mandates that if AC13 holds and AC14 does not hold for
6.2. THE UMLRT-TO-KILTERA CASE STUDY

Figure 6.13: AtomicContracts AC13, AC14, AC15, and AC16 that are used to express SS3 (Table 6.4) as \((AC13 \land \neg tc \rightarrow AC14) \implies tc (AC15 \land \neg tc \rightarrow AC16)\).

a specific Inst element, then AC15 should hold and AC16 should not hold for the same Inst element.

We demonstrate the formulation of pattern contracts (e.g., PP1, PP2, PP3, PP4, and PP5 in Table 6.4) in our property prover by showing the formulation of PP1 in Fig. 6.14 as an example. PP1 mandates that if a sibling Transition has a Trigger with a Signal element in the input model (as shown in the precondition of Fig. 6.14), then a ListenBranch will be connected to the Inst corresponding to the input Transition (with four Name elements passed as channelNames) in the output model (as shown in the postcondition of Fig. 6.14). As explained in Section 5.1, using a traceability link
6.2. THE UMLRT-TO-KILTERA CASE STUDY

![AtomicContract AC17 diagram]

Figure 6.14: AtomicContract AC17 that is used to express PP1 (Table 6.4).

in Fig. 6.14 mandates that PP1 will only match Inst elements that were previously created from Transition elements.

Rule reachability is verified in a different manner. During the generation of the set of path conditions, we store the rules that are used to produce each path condition. Thus, reachability of a specific rule is verified by checking if that rule was used to generate at least one path condition.

6.2.6 Verification Results

We used our symbolic model transformation property prover to verify the properties shown in Table 6.4 for our UML-RT-to-Kiltera model transformation shown in Fig. 6.7. The transformation was found to violate MM11, SS2, and RR15 i.e., the transformation can generate the following bugs: (i) an output where a Par is associated to less than two Procs (violating MM11), (ii) an output where an Inst named Dispatcher is created but a corresponding ProcDef named Dispatcher is not created (violating SS2), and/or (iii) no input model can trigger rule ‘MapNesting’ in layer 6, i.e., the rule is a dead rule and is not reachable through any possible execution
After examining the generated counter examples, we identified two bugs in the transformation shown in Fig. 6.7.

1. Execution of the rule ‘ExitPoint2ProcDef’ (layer 3) does not mandate execution of the rule ‘MapExitWithTrans’ (layer 5) in the buggy transformation (Fig. 6.7). For example, an input model containing a composite State with an ExitPoint where the ExitPoint has no outgoing Transitions will cause rule ‘ExitPoint2ProcDef’ (layer 3) to execute (but not rule ‘MapExitWithTrans’ in layer 5). This will result in an output containing a Par associated to one Trigger (which inherits from Proc), violating MM11.

2. Execution of the rule ‘MapCompositeState’ (layer 2) does not mandate execution of the rule ‘State2Dispatcher’ (layer 3) in the buggy transformation (Fig. 6.7). For example, an input model containing a composite State without an initial Transition will cause rule ‘MapCompositeState’ (layer 2) to execute (but not rule ‘State2Dispatcher’ in layer 3). This will result in an output containing an Inst named Dispatcher, but not a ProcDef named Dispatcher, violating SS2.

The violation of RR15 (i.e., the reachability of rule ‘MapNesting’ in layer 6 of Fig. 6.7) is not a bug; it is due to the fact that our property prover assumes that each rule executes once (i.e., each rule has one match) during path condition generation. Rule ‘MapNesting’ in layer 6 will execute only if the input model has at least two State elements, i.e., if the rule ‘State2ProcDef’ in layer 1 executes at least twice (or finds at least two matching State elements). Taking this limitation of our property prover into consideration and to verify the reachability of rule ‘MapNesting’, we add to layer 1 of the transformation a duplicate of rule ‘State2ProcDef’ to emulate the fact
that the rule executed twice. Reverifying $RR15$ after adding the duplicate of rule ‘State2ProcDef’ in layer 1 returns $true$, meaning that the rule ‘MapNesting’ (layer 6) is reachable if the rule ‘State2ProcDef’ (layer 1) executes twice.

We fixed the transformation to address the above mentioned bugs. Fig. 6.15 shows the updated transformation. The updated transformation (Fig. 6.15) differs from the buggy transformation (Fig. 6.7) in two ways:

1. The two rules ‘ExitPoint2ProcDef’ (layer 3) and ‘MapExitWithTrans’ (layer 5) in the buggy transformation (Fig. 6.7) were replaced with one merged rule ‘MapExitWithTrans’ (layer 5) in the updated transformation (Fig. 6.15). This change will generate two $Proc$s associated to one $Par$ in the same rule, as opposed to generating the two associated $Proc$s in two separate rules that will not necessarily execute together (as in the buggy transformation in Fig. 6.7).

2. The match model of the rule ‘MapCompositeState’ (layer 2) has been updated to include the $Exists$ match elements in the match model of the rule ‘State2Dispatcher’ (layer 3). This change should ensure that execution of one rule mandates the execution of the other rule, too. Thus, if an $Inst$ named $Dispatcher$ is created by the rule ‘MapCompositeState’, then a $ProcDef$ named $Dispatcher$ will also be created by rule ‘State2Dispatcher’ (satisfying $SS2$).

The properties in Table 6.4 were reverified on the updated UML-RT-to-Kiltera model transformation in Fig. 6.15, and verification of all the properties returned $true$ (i.e., the DSLTrans implementation of our updated UML-RT-to-Kiltera model transformation in Fig. 6.15 will always satisfy the properties shown in Table 6.4).

To assess the performance of our property prover, we measured the time taken to generate the path conditions and the time taken to verify the properties (shown in
Figure 6.15: Layers 1-3 of the updated UML-RT-to-Kiltera transformation implementation (figure continued in the next page).
Figure 6.15: Layers 4-6 of the updated UML-RT-to-Kiltera transformation implementation.
Table 6.4) for the updated UML-RT-to-Kiltera model transformation after fixing the bugs (Fig. 6.15). Our property prover took on average 13.26 seconds to generate 57 path conditions for the updated transformation in Fig. 6.15. Table 6.5 shows the time taken (in seconds) to verify the multiplicity invariants listed in Table 6.4, Table 6.6 shows the time taken (in seconds) to verify the syntactic invariants and the pattern contracts listed in Table 6.4, and Table 6.7 shows the time taken (in seconds) to verify the reachability properties listed in Table 6.4. In Table 6.7, we have NA as the time taken to verify RR5 (i.e., the reachability of rule ‘ExitPoint2ProcDef’ in layer 3), since this rule has been removed for the fixed transformation (as explained previously and as shown in Fig. 6.15). For verifying RR15 (i.e., the reachability of the rule ‘MapNesting’ in layer 6), we duplicated the rule ‘State2ProcDef’ in layer 1 (as previously described in this section). Adding a duplicate rule in the first layer resulted in the generation of 172 path conditions (in 77.40 seconds), and verifying RR15 in that case took $2.70 \times 10^{-4}$ seconds (as shown in Table 6.7). We do not include the time taken for path condition generation in Tables 6.5, 6.6, and 6.7 since it is performed once for the model transformation.

<table>
<thead>
<tr>
<th>Property</th>
<th>MM1</th>
<th>MM2</th>
<th>MM3</th>
<th>MM4</th>
<th>MM5</th>
<th>MM6</th>
<th>MM7</th>
<th>MM8</th>
<th>MM9</th>
<th>MM10</th>
<th>MM11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification Time</td>
<td>1.59</td>
<td>1.23</td>
<td>0.58</td>
<td>0.58</td>
<td>0.95</td>
<td>0.98</td>
<td>0.90</td>
<td>1.33</td>
<td>1.39</td>
<td>1.18</td>
<td>1.13</td>
</tr>
</tbody>
</table>

Table 6.5: Time taken (in seconds) to verify the multiplicity invariants shown in Table 6.4 using our property prover.

<table>
<thead>
<tr>
<th>Property</th>
<th>SS1</th>
<th>SS2</th>
<th>SS3</th>
<th>PP1</th>
<th>PP2</th>
<th>PP3</th>
<th>PP4</th>
<th>PP5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification Time</td>
<td>5.50</td>
<td>5.38</td>
<td>241.41</td>
<td>12.72</td>
<td>11.07</td>
<td>21.61</td>
<td>10.68</td>
<td>3.15</td>
</tr>
</tbody>
</table>

Table 6.6: Time taken (in seconds) to verify the syntactic invariants and the pattern contracts shown in Table 6.4 using our property prover.
Table 6.7: Time taken (in seconds) to verify the reachability properties shown in Table 6.4 using our property prover.

As shown in Table 6.5, the times taken to verify multiplicity invariants were very short (ranging from 0.58 seconds for $MM3$ to a maximum of 1.59 seconds for $MM1$).

The verification times for syntactic invariants and pattern contracts were longer than the verification times of multiplicity invariants, as shown in Table 6.6. We attribute these long verification times to the time required to perform graph matching (subgraph isomorphism) during property verification, i.e., the more elements to be matched in a property, the longer is the verification time. For example, verifying $SS1$ and $SS2$ took a bit more than 5 seconds each, while verifying the pattern contracts took a maximum verification time of 21.61 seconds (for $PP3$). Looking at the $AtomicContracts$ used to express the syntactic invariants and the pattern contracts (shown in Appendix C), we note that these $AtomicContracts$ have more elements (and attributes) to match when compared to the $AtomicContracts$ used to express any of the multiplicity invariants. Of all the properties being verified, $SS3$ took an exceptionally long time to verify (241.41 seconds). Again, this is attributed to the fact that $SS3$ is the only property expressed using four $AtomicContracts$ (shown in Fig. 6.13), where each $AtomicContract$ will have to be matched on top of all the 57 path conditions. Moreover, each of the four $AtomicContracts$ used to express $SS3$ shown in Fig. 6.13 has more elements (and attributes) when compared to $AtomicContracts$ used to express other properties (such as the multiplicity invariants).
6.3. DISCUSSION

Verifying reachability for all the rules took negligible time (as shown in Table 6.7), with a maximum of $2.70 \times 10^{-4}$ seconds (for RR15). This is due to the fact that reachability was not checked using graph matching, as explained in Section 6.2.5. During path condition generation, we store the rules that are used to produce each path condition and reachability of a rule is verified by checking if that rule was used to produce at least one path condition.

Since proving all the properties in Table 6.4 except for rule reachability requires graph matching and since proving rule reachability took a negligible amount of time, the performance of our property prover is proportional to the performance of the graph matching carried out by T-core (the underlying language used to implement our property prover, as explained in Section 5.2). The graph matching algorithm of T-core is a variation of the VF2 graph matching algorithm [147] whose time complexity is $\Theta(N!N)$ (in the worst case) for a graph with $N$ nodes [40].

6.3 Discussion

Based on the two case studies that we conducted and reported on in Section 6.1 and Section 6.2, we discuss the strengths and limitations of our symbolic model transformation property prover.

6.3.1 Strengths of the Symbolic Model Transformation Property Prover

As mentioned at the beginning of Chapter 5, our property prover was initially intended to address many of the shortcomings of existing model transformation verification approaches and tools. Based on the two experiments that we conducted on the GM-to-AUTOSAR model transformation and the UML-RT-to-Kiltera model
transformation, we conclude that our property prover has achieved its initial purpose and has the following strengths in comparison with the state-of-the-art model transformation verification approaches.

First, our symbolic model transformation property prover is not restricted to verifying a certain class of properties (e.g., forbidden patterns [23]). Instead, our prover can conclusively verify (unlike [13]) a wide range of property types such as properties related to the transformation’s output only (e.g., multiplicity invariants of different arities, security invariants, syntactic invariants), properties that relate the transformation’s input and output (e.g., pattern contracts), and properties of the transformation rules (e.g., rule reachability). Besides reasoning about elements of a certain type, properties can also reason about attribute values (e.g., SS1 in Fig. 6.12 and SS2 in Fig. C.13). Other useful properties such as termination and confluence are already guaranteed by construction of the DSLTrans language [18].

Second, graphical model transformation languages (e.g., the DSLTrans language used by our property prover) are notationally consistent with the philosophy of models perceived as graphs, and hence, are easy to understand by users (as opposed to verification techniques based on textual formalisms such as Maude [152, 106]). In the case of our property prover, the model transformation rules and the properties are expressed using the same constructs, making the verification tool and its underlying language consistent and easy to learn. Further, our property prover generates the path conditions and the counter examples in a graphical format, making debugging the transformation and its verification results a relatively easy task.

Third, we claim that the input independent verification technique used by our symbolic model transformation property prover is a major strength of our prover in
6.3. DISCUSSION

comparison with many other state-of-the-art verification tools that verify a transformation for a specific input (e.g., Henshin [10] and AGG [148, 129]). For a transformation that is intended to transform numerous models, it is crucial to ensure that the transformation preserves certain properties without having to reverify these properties each time the transformation maps a new input model. If the transformation is to be used for embedded or safety critical systems, avoiding the verification of transformation properties for every input becomes a requirement to ensure execution in a timely fashion.

Fourth, several studies proposed model transformation verification techniques and tools, but rarely did such studies demonstrate the tools’ scalability and applicability to different transformation types. We demonstrated our property prover on two different types of transformations of varying sizes. The GM-to-AUTOSAR model transformation is an industrial, model-to-model transformation of structural models. Whereas the UML-RT-to-Kiltera model transformation is an academic, model-to-text transformation of behavioural models. Moreover, the two transformations have different intents [4, 89] (i.e., the GM-to-AUTOSAR transformation is a migration transformation and the UML-RT-to-Kiltera transformation is an analysis transformation). For each of the two transformations, we discussed the verification results and how the prover helped uncover bugs, and we demonstrated the times taken to verify different kinds of properties. As shown for the GM-to-AUTOSAR transformation (Table 6.2) and for the UML-RT-to-Kiltera transformation (Tables 6.5, 6.6, and 6.7), the prover can verify different property types in a relatively short time. While in one case our symbolic model transformation property prover took 241.41 seconds to verify a property (as shown in Table 6.6), we believe that the execution time is acceptable given
that this verification will be done once over the lifetime of the transformation. We also discussed in Section 6.1.3 how our property prover scales well in comparison with the tool we experimented with in Section 4.2.

Finally, when verifying model transformations using intermediate representations of the transformation (e.g., path conditions or a transformation model [35, 34]), it is crucial to prove the soundness and completeness of the intermediate representations. A sound transformation representation is one that does not have any erroneous representation that does not truly reflect a possible execution of the transformation. A complete transformation representation is one that captures all possible transformation executions. The soundness and completeness of the path conditions generated by our property prover (and that are used to verify properties) have been previously proved by our collaborators [91]. This is in contrast with state-of-the-art verification techniques that translate transformations into an intermediate formalism and verify properties on the translated transformation without proving the soundness of the translated transformation [35, 36, 6, 17, 152, 106]. Moreover, such approaches should translate the verification result back to the original formalism for comprehension. Other studies proposed incomplete techniques that are restricted to a scope [35] or that do not guarantee that the transformation is fault-free, e.g., testing.

6.3.2 Limitations of the Symbolic Model Transformation Property Prover

We identify three limitations of our symbolic model transformation property prover.

First, while our symbolic model transformation property prover can reason about attribute values (e.g., property SS1 in Table 6.4 reasons about \textit{Inst} and \textit{ProcDef} elements whose \textit{name} attributes equals \textit{Handler}), a more general purpose action
language is needed that facilitates reasoning about some variable attribute value without having to specify that value. Such an action language should allow us, for example, to express \textit{SS1} and \textit{SS2} (Table 6.4) as one generalized property, as follows: “If an \textit{Inst} with \textit{name X} is created, then a \textit{ProcDef} with \textit{name X} is also created”, where \textit{X} can be any string value. Further, \textit{SS3} can be expressed as a more generalized property using such an action language, as follows: “If an \textit{Inst} with \textit{name X} is associated to \textit{M Name} elements, then a \textit{ProcDef} with \textit{name X} is also created and is associated to \textit{M Name} elements” where \textit{X} can be any string value and \textit{M} can be any integer value.

Second, while negative application conditions (NACs) are expressible in DSLTrans, our current implementation of the symbolic model transformation property prover cannot generate path conditions or verify properties for transformations with rules having NACs. Incorporating NACs to our symbolic model transformation property prover will enable verifying more properties for a wider range of transformations, where both the properties and the transformation rules can have NACs.

Finally, our property prover currently assumes that each rule executes once during path condition generation. Thus, proving properties that rely on a rule executing more than once will return false, even if the property holds for the transformation. In Section 6.2.6 for example, we showed that the reachability of rule ‘MapNesting’ in layer 6 (i.e., \textit{RR15}) initially returned false since execution of the rule ‘MapNesting’ requires the execution of rule ‘State2ProcDef’ in layer 1 at least twice. While the first two limitations can be addressed in the future, this last limitation is due to the nature of the property prover (i.e., being independent of the input and independent of the number of times that a rule can execute). Thus, while an automated solution to this
limitation might not be feasible, our current workaround was to reverify $RR15$ after duplicating the rule ‘State2ProcDef’ in layer 1. In other words, to verify properties that should hold if a certain rule runs more than once, we can add duplicates of that rule to the transformation before verifying the property.

6.4 Summary

In this chapter, we demonstrated the usefulness of our symbolic model transformation property prover (discussed in Chapter 5) by conducting and reporting on two case studies. In the first case study, we used our property prover to verify three kinds of properties for the GM-to-AUTOSAR transformation: multiplicity invariants, security invariants, and pattern contracts. In the second case study, we used our property prover to verify four kinds of properties for the UML-RT-to-Kiltera transformation: multiplicity invariants, syntactic invariants, pattern contracts, and rule reachability.

The two model transformations differ in their size and nature, which implies the wide applicability of our property prover to different transformations. For example, the GM-to-AUTOSAR transformation is an industrial, model-to-model transformation of structural models. Whereas the UML-RT-to-Kiltera model transformation is an academic, model-to-text transformation of behavioural models. Further, the two transformations differ in their intents [4, 89], where the GM-to-AUTOSAR transformation can be classified as a migration transformation (i.e., the transformation is developed to migrate a class of models conforming to one metamodel to an equivalent class of models conforming to another metamodel with similar semantics as the first metamodel) and the UML-RT-to-Kiltera transformation can be classified as an analysis transformation (i.e., the transformation is developed to transform a class of
models into another class of models conforming to a metamodel with well-defined semantics that enables analysis of the transformed models).

In the two case studies, we discussed the objective of the transformation, the source and target metamodels and the required mapping rules between them, and the implementation details of the transformations. Moreover, we proposed properties of interest for each transformation, we demonstrated how these properties were formulated in our property prover, and we discussed the verification and performance results. Finally, we discussed the strengths and limitations of our symbolic model transformation property prover based on the two conducted case studies and based on our experience with the state-of-the-art tools (Chapter 4).

These two case studies demonstrate that our property prover can be used to verify a range of different property types for model transformations of different kinds and sizes with acceptable run times. More case studies for model transformations of bigger sizes is required before we can confirm the scalability of our property prover.
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we investigated the problem of verifying several property types for graph-based model transformations in an automatic and scalable fashion using symbolic execution techniques. We started off by reviewing the state-of-the-art in model transformation verification to get a broader view of existing techniques (Chapter 2, [137, 5, 3, 138, 4, 89]). Then, we developed an industrial model transformation (Chapter 3, [141]) which we used to evaluate two existing model transformation verification tools (Chapter 4, [136, 142]). The conducted literature review and the experimentation with the existing tools gave us a better understanding of the limitations of available model transformation verification techniques that need to be addressed. In an attempt to overcome such limitations, we redeveloped and extended a symbolic model transformation property prover for the DSLTrans model transformation language (Chapter 5, [140, 139]). We further evaluated our property prover using two model transformations of different natures (one of which is our industrial model transformation), and we reported on the verification and performance results, as well as
the strengths and limitations of our property prover (Chapter 6, [140, 139]).

We claim that our property prover is practical to use since it is based on an intuitive graphical model transformation language and has been demonstrated on two transformations of varying natures and sizes. For the two transformations, our property prover was able to conclusively verify many property types (e.g., multiplicity invariants of different arities, security invariants, syntactic invariants, pattern contracts, and rule reachability) in relatively short times. Verification of certain properties (e.g., SS3 in Table 6.6) took longer (i.e., more than 4 minutes) than the verification of the other properties. We attributed this longer verification time to the number of elements and attributes in a property, i.e., since graph matching is performed between the property of interest and each of the generated path conditions, more elements and attributes to match in the property imply a longer verification time. We believe that such longer verification times are acceptable given that our property prover is input-independent and verification of properties is performed once over the lifetime of the transformation. Further, our property prover helped in uncovering and fixing bugs in the two model transformations. Finally, the soundness and completeness of the generated path conditions gives a higher confidence in the verification results in comparison with state-of-the-art tools.

Like any other model transformation verification approach and as noted at the beginning of Chapter 5, there will always be a trade off between the expressiveness of model transformation languages and the verifiability of these languages. The limited expressiveness of DSLTrans makes it possible to perform unbounded, formal verification of model transformation properties. Based on our experience in developing the two model transformations used as case studies in Chapter 6, we can claim that
7.2. FUTURE WORK

DSLTrans can be used to implement non-trivial transformations (as long as they do not have unbounded loops). Thus, we believe that DSLTrans maintains a good balance between the two factors (i.e., expressiveness and verifiability), which makes it a good candidate for model transformation verification tools.

This thesis adds to the state of the art (Section 2) and is useful to transformation verification research in general. We provide some evidence for our property prover’s scalability and usefulness. Thus, we motivate researchers to adopt our symbolic model transformation property prover. Moreover, users of languages other than DSLTrans can benefit from our study in two ways: (1) the study can be used as a guide to develop input-independent verification tools for any model transformation language; (2) higher order transformations (HOTs) can be developed to convert model transformations in other languages to DSLTrans to enable using our property prover. To develop such HOTs, more research has to be conducted to understand what class of model transformations can be translated to DSLTrans.

7.2 Future Work

This research can be extended in several ways to improve the capabilities of our symbolic model transformation property prover.

First, our current implementation of our symbolic model transformation property prover does not have a graphical user interface. Instead, separate files of graph transformation rules are built and scheduled in a Python module to represent the transformation being verified. Moreover, properties are created as individual files of graph properties, and these files are used as arguments for the property verification code. The next natural step would be to develop a graphical user interface to allow end
users to develop transformations and formulate properties in the same development environment, without having to manipulate the code of the property prover.

Second, the limitations of our current implementation of our property prover (discussed in Section 6.3.2) should be addressed.

- Developing a more flexible action language that reasons about attribute values will allow formulating properties that are less verbose. As discussed in Section 6.3.2, such an action language will allow us to formulate, for example, properties $SS1$ and $SS2$ (Table 6.4) as one generalized property (or propositional formula) that uses smaller number of $AtomicContracts$.

- Incorporating negative application conditions (NACs) in our prover will facilitate verifying more property types for a bigger variety of transformations, where both the transformation rules and the properties can contain NACs.

Third, due to our recent interest in software product lines [51], we plan to investigate how properties of product lines can be extrapolated from model transformation invariants and contracts (like the invariants and contracts we proposed for the GM-to-AUTOSAR transformation in Table 4.2 and the UML-RT-to-Kiltera transformation in Table 6.4), and how can we facilitate verifying such properties in our property prover. We also plan to investigate verifying transformation-related properties (e.g., rule reachability in Table 6.4) of “lifted” transformations [130] that operate on product lines instead of individual models. In this case, verification will be used to check if properties of the original transformation are preserved in the lifted transformation, i.e., verification will be used to ensure that the transformation lifting was done correctly without violating any of the properties of the original transformation.
Bibliography


[27] Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In Marsha Chechik and Martin


[62] Esther Guerra, Juan De Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel,


[129] Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0–New Features for Specifying and Analyzing Algebraic Graph Transformations. In Andy Schürr,


Appendix A

Formalizations

We briefly overview the formalizations used by the studies surveyed in this thesis.

A.1 Graph Rewriting Systems

A graph rewriting system [128, 26] is a formal model in which the static states of a system are represented as graphs, and the behavior and evolution of a system are represented as graph rewriting rules on those graphs [126]. The manipulated graphs can be typed and attributed, where the graph rewriting rules must be type-preserving and must handle rewriting the attributes. The terms graph rewriting systems and graph transformation systems were used interchangeably in the literature. In this thesis, we use the term graph rewriting systems to unify the used terminology.

A graph rewriting system [128, 26] is composed of one or more graph rewriting rules. A graph rewriting rule has a left-hand side (LHS), a right-hand side (RHS) and optional negative application conditions (NACs). The LHS is a graph pattern to look for in the host graph, the RHS is the graph pattern to create if the LHS was found in the host graph, and the NACs are graph constraints which prohibit the existence of certain patterns in the host graph. The execution of a graph rewriting
A.1. GRAPH REWRITING SYSTEMS

rule $r$ on a host graph $G$ is carried out as follows: (i) find a match for the LHS of $r$ in $G$, (ii) check whether the match found satisfies the NACs of $r$, (iii) remove all the graph elements from $G$ which have an image in the LHS but not in the RHS, and (iv) create new graph elements in $G$ for all elements that have an image in the RHS but not in the LHS. Thus, the LHS and the NACs are the preconditions of the graph rewriting rule while the RHS is the postcondition of the graph rewriting rule. Two graph rewriting approaches were proposed in the literature: the Double Pushout (DPO) approach and the Single Pushout (SPO) approach [128].

Several variants of graph rewriting systems were proposed and used in the literature. For example, Hypergraph rewriting systems are generalizations of graph rewriting systems where an edge can connect more than two nodes [128]. High level replacement units (HLRUs) are generalizations of graph rewriting systems (i.e., they allow manipulating different kinds of graphs) with a control mechanism for rule applications, e.g., allowing sequential application of rules, or applying just a single rule.

A graph rewriting system and an initial host graph is referred to as a graph grammar [128]. A graph transition system generated from a graph grammar is the state space of the graph grammar where nodes represent intermediate graphs, and transitions represent possible graph rewriting rule applications [123].

Graph rewriting systems have been traditionally used to design terminating and confluent transformations. Graph rewriting systems are considered to have limited analysis capabilities in comparison to other formalizations especially if they manipulate attributed graphs [127].
A.2 Triple Graph Grammars (TGGs)

One major disadvantage of graph rewriting systems is that they are usually restricted to performing in-place graph rewriting between graph instances conforming to the same graph. Thus, they cannot be easily used to track traceability links between input and output graph instance elements. A Triple Graph Grammar (TGG) [132] specification is a declarative definition of a bidirectional graph rewriting system. TGGs overcome the disadvantage of graph rewriting systems by using correspondence graphs or metamodels that maintain m-to-n relationships between the input and output graph instance elements, hence maintaining consistency between them. One TGG rule is composed of three rule components; a left rule, a right rule and a correspondence rule, each responsible for matching and rewriting the corresponding graphs. Each rule component has a LHS and a RHS. Thus, the notion of triple captures the input, output and correspondence graph instances that are rewritten in parallel in any TGG rule.

Using one TGG rule, one can perform source-to-target transformation, target-to-source transformation, or correspondence analysis (i.e., analyzing the relationships between the input and output graph instance elements) without having to define three separate graph rewriting rules for each operation.

A.3 Petri Nets

Petri Nets [111] are formal models of information flow and control in systems, especially asynchronous and concurrent systems. Petri Nets are composed of four basic elements: places, transitions, edges and (optional) weights [155].

Places are represented graphically as hollow circles and are used to represent
A.3. PETRI NETS

states of a system. Places may contain one or more tokens, represented graphically by black solid dots within the places. The distribution of tokens in different places of a Petri Net is called the marking of the Petri Net. The initial marking of a Petri Net refers to the initial distribution of tokens in a Petri Net. A transition is represented graphically as a horizontal bar and represents a possible change in the system state. A transition can be connected to one or more input places by incoming edges. Similarly, a transition can be connected to one or more output places by outgoing edges. A transition is said to be enabled (i.e., the transition may fire) if each of its input places contains at least as many tokens as specified by the weights associated with the incoming edges. If no weights are specified, then a weight of one is assumed. If a transition is enabled and fires, the marking of a Petri Net changes. This is done by removing tokens from the input places of the transition and producing tokens in the output places of the transition. The number of tokens removed and produced by the firing of a transition is determined by the weights associated with the edges. Timed Transition Petri Nets (TTPNs) are one variant of Petri Nets that were used by one of the surveyed studies [42]. TTPNs are similar to regular Petri Nets with the exception of associating a delay with each transition, such that the transition has to be enabled for a number of time units equal to the delay before firing.

Many studies transformed formal models such as graph rewriting systems to some variant of Petri Nets due to their strong support for analysis (e.g., correctness analysis, dependability analysis, performance analysis [49] and termination analysis [155]). For example, Varró et al. [155] abstracted graph rewriting systems into Petri Nets to perform termination analysis. Node and edge types were represented as places; rules were represented as transitions; the LHS and the RHS of a rule were represented as
weighted arcs between the place of the corresponding elements type and the transition of the corresponding rule; and the input graph instance determined the initial marking of the Petri Net.

A.4 Alloy

Alloy [73] is a declarative modelling language based on first order relational logic with well-defined semantics. Signatures, relations, facts and predicates are used to specify a model in Alloy. Signatures represent the entities of a system and relations represent the relations between such entities. Facts and predicates specify constraints on signatures and relations.

Alloy comes with an analyzer, the Alloy analyzer [72], that uses constraint solvers to automatically analyze Alloy models. The Alloy analyzer supports two kinds of analysis: consistency checking and assertion checking. Consistency checking verifies that the specified Alloy model is consistent by generating a random model instance that conforms to the Alloy model specification. Assertion checking ensures that the Alloy model satisfies some assertions or constraints. If assertion checking fails, the Alloy analyzer generates a counter example or an instance model that violates one or more assertions. Although Alloy is a modelling language, it has been used to specify model transformations (e.g., [95, 6]), thus allowing researchers to take advantage of the Alloy analyzer.

A.5 Maude

Maude [39] is a language and an engine that supports Membership Equational Logic (MEL) [31] specifications and rewriting logic specifications. A Maude specification
of a system can have functional modules and system modules. A functional module specifies membership equational theories that describe possible states of the modeled system. A functional module also uses equations as simplification rules to find unique and simplified forms of terms conforming to the membership equational theories. System modules specify rewrite theories that rewrite terms conforming to membership equational theories.

Maude supports three kinds of analysis: simulation, reachability analysis and model checking linear temporal logic (LTL) properties. Simulation is the execution of a Maude system specification. Reachability analysis builds the state space of a Maude system specification to look for deadlocks (i.e., states on which no further rewrite may take place) and to prove or disprove system invariants by generating counter examples. Model checking LTL properties involves checking whether every possible behavior of a Maude system specification, starting from an initial model, satisfies a LTL property. LTL properties (i.e., properties that can be specified in linear temporal logic) take the form of safety properties (i.e., ensuring that something bad never happens) or liveness properties (ensuring that something good eventually happens)\(^1\) [39]. If an LTL property is violated, Maude generates a counter example too.

Maude has been used to represent model transformations and their manipulated metamodels and models due to several reasons besides its support for analysis. Maude’s rewriting logic preserves conformance of output models to metamodel constraints. Maude also manipulates models in a consistent way, e.g., deleting orphan nodes and dangling edges.

\(^1\)In the rest of this thesis, we refer to safety properties and liveness properties as LTL properties.
Appendix B

ATL Pragmatics

We summarize the main concepts of ATL and we shed light on some of ATL’s elements and their use. Additional resources such as the ATL manual [48] and the ATL Zoo [56] are useful in understanding more about ATL.

In ATL, a model transformation is defined as a set of rules and helpers. Rules specify the creation of output model elements. Helpers are used to modularize a transformation. ATL defines four types of rules and two types of declarative helpers.

**Rule Types.** The four types of rules are matched rules, lazy rules, unique lazy rules, and called rules. A matched rule specifies the source pattern to match in the input model and the corresponding target pattern to create in the output model. Matched rules are executed in the order of their specification and are automatically executed once for each matching pattern. A lazy rule is executed only when called and can be called multiple times for the same matching pattern. A unique lazy rule is executed only when called and can be called at most once for any matching pattern. A called rule is a parameterized rule that is executed only when called and creates a target pattern without matching any source patterns. All rule types have an optional imperative code block that can be used to specify complicated functionality.
Matched rules are suitable for automatic detection of all matching patterns in the input model and creation of their corresponding target patterns. Lazy rules and unique lazy rules are suitable for selective pattern matching, with consideration of the number of times these rules should be run. Called rules are suitable for creating output model elements that do not match any input model elements.

**Helper Types.** The two types of helpers are functional helpers and attribute helpers. A functional helper is a parametric function and is evaluated each time it is invoked. An attribute helper is a non-parametric function and is evaluated only the first time it is invoked. Thus, an attribute helper is more efficient to implement a non-parametric functionality. Otherwise, a functional helper can implement a parametric functionality.

**Model Transformation Specification.** Similarly to source transformation languages, there are two approaches to specifying transformations in ATL: specifying the transformation as one large rule, or modularizing the transformation using smaller rules and helpers. As in any other transformation language, the two approaches present trade-offs between ease of implementation and efficiency. Building one large rule makes all variables accessible throughout the transformation, so the developer need not worry about the ordering of rules in the transformation specification. However, this approach makes the transformation difficult to maintain and less readable. Modularizing the transformation makes the transformation easier to debug and maintain. However, the developer has to ensure that the rules are specified in an order consistent with the dependencies among rules.
Appendix C

Formulated Properties for the UML-RT-to-Kiltera Model Transformation

We demonstrate the formulation of all the properties (Table 6.4) proposed for the UML-RT-to-Kiltera model transformation in our property prover.

C.1 Formulation of Multiplicity Invariants

‘1..1’ Multiplicity Invariants: Using the AtomicContracts shown in Fig. C.1, the multiplicity invariant MM1 can be expressed as $AC1 \implies_{tc} (AC2 \land_{tc} \neg_{tc} AC3)$. Using the AtomicContracts shown in Fig. C.2, the multiplicity invariant MM2 can be expressed as $AC18 \implies_{tc} (AC19 \land_{tc} \neg_{tc} AC20)$. Using the AtomicContracts shown in Fig. C.3, the multiplicity invariant MM3 can be expressed as $AC21 \implies_{tc} (AC22 \land_{tc} \neg_{tc} AC23)$. Using the AtomicContracts shown in Fig. C.4, the multiplicity invariant MM4 can be expressed as $AC24 \implies_{tc} (AC25 \land_{tc} \neg_{tc} AC26)$.

‘1..*’ Multiplicity Invariants: Using the AtomicContracts shown in Fig. C.5, the multiplicity invariant MM5 can be expressed as $AC4 \implies_{tc} AC5$. Using the AtomicContracts shown in Fig. C.6, the multiplicity invariant MM6 can be expressed
C.1. FORMULATION OF MULTIPLICITY INVARIANTS

Figure C.1: AtomicContracts AC1, AC2, and AC3 that are used to express MM1 (Table 6.4) as AC1 \xrightarrow{tc} (AC2 \land \neg tc AC3).

Figure C.2: AtomicContracts AC18, AC19, and AC20 that are used to express MM2 (Table 6.4) as AC18 \xrightarrow{tc} (AC19 \land \neg tc AC20).

as AC27 \xrightarrow{tc} AC28. Using the AtomicContracts shown in Fig. C.7, the multiplicity invariant MM7 can be expressed as AC29 \xrightarrow{tc} AC30.

‘0..1’ Multiplicity Invariants: Using the AtomicContracts shown in Fig. C.8, the multiplicity invariant MM8 can be expressed as AC6 \xrightarrow{tc} ((AC7 \land \neg tc AC8) \lor tc (\neg tc AC7)). Using the AtomicContracts shown in Fig. C.9, the multiplicity invariant MM9 can be expressed as AC33 \xrightarrow{tc} ((AC34 \land \neg tc AC35) \lor tc (\neg tc AC34)). Using the AtomicContracts shown in Fig. C.10, the multiplicity invariant MM10 can be expressed as AC36 \xrightarrow{tc} ((AC37 \land \neg tc AC38) \lor tc (\neg tc AC37)).

‘2..*’ Multiplicity Invariants: Using the AtomicContracts shown in Fig. C.11, the multiplicity invariant MM11 can be expressed as AC9 \xrightarrow{tc} AC10.
C.2 Formulation of Syntactic Invariants

Using the AtomicContracts shown in Fig. C.12, the syntactic invariant SS1 can be expressed as $AC11 \implies_{tc} AC12$. Using the AtomicContracts shown in Fig. C.13, the syntactic invariant SS2 can be expressed as $AC39 \implies_{tc} AC40$. Using the AtomicContracts shown in Fig. C.14, the syntactic invariant SS3 can be expressed as $(AC13 \land_{tc} \neg_{tc} AC14) \implies_{tc} (AC15 \land_{tc} \neg_{tc} AC16)$. 

\[ \text{Figure C.3: AtomicContracts } AC21, AC22, \text{ and } AC23 \text{ that are used to express } MM3 \text{ (Table 6.4) as } AC21 \implies_{tc} (AC22 \land_{tc} \neg_{tc} AC23). \]

\[ \text{Figure C.4: AtomicContracts } AC24, AC25, \text{ and } AC26 \text{ that are used to express } MM4 \text{ (Table 6.4) as } AC24 \implies_{tc} (AC25 \land_{tc} \neg_{tc} AC26). \]
C.3. FORMULATION OF PATTERN CONTRACTS

The pattern contract \(PP1\) can be expressed using the \(AtomicContract\ AC17\) shown in Fig. C.15. The pattern contract \(PP2\) can be expressed using the \(AtomicContract\ AC17\) shown in Fig. C.15.
Figure C.8: *AtomicContracts AC6, AC7, and AC8* that are used to express MM8 (Table 6.4) as \( AC6 \Rightarrow tc \left( (AC7 \land tc \neg tc AC8) \lor tc \left( \neg tc AC7 \right) \right) \).

Figure C.9: *AtomicContracts AC33, AC34, and AC35* that are used to express MM9 (Table 6.4) as \( AC33 \Rightarrow tc \left( (AC34 \land tc \neg tc AC35) \lor tc \left( \neg tc AC34 \right) \right) \).

Figure C.10: *AtomicContracts AC36, AC37, and AC38* that are used to express MM10 (Table 6.4) as \( AC36 \Rightarrow tc \left( (AC37 \land tc \neg tc AC38) \lor tc \left( \neg tc AC37 \right) \right) \).

AC41 shown in Fig. C.16. The pattern contract PP3 can be expressed using the *AtomicContract AC42* shown in Fig. C.17. The pattern contract PP4 can be expressed using the *AtomicContract AC43* shown in Fig. C.18. The pattern contract PP5 can be expressed using the *AtomicContract AC44* shown in Fig. C.19.
C.3. FORMULATION OF PATTERN CONTRACTS

Figure C.11: AtomicContracts AC9 and AC10 that are used to express MM1 (Table 6.4) as $AC9 \rightarrow_{tc} AC10$.

Figure C.12: AtomicContracts AC11 and AC12 that are used to express SS1 (Table 6.4) as $AC11 \rightarrow_{tc} AC12$.

Figure C.13: AtomicContracts AC39 and AC40 that are used to express SS2 (Table 6.4) as $AC39 \rightarrow_{tc} AC40$. 
Figure C.14: AtomicContracts AC13, AC14, AC15, and AC16 that are used to express SS3 (Table 6.4) as (AC13∧¬tcAC14) ⇒tc (AC15∧¬tcAC16).

Figure C.15: AtomicContract AC17 that is used to express PP1 (Table 6.4).
Figure C.16: AtomicContract AC\textsubscript{41} that is used to express PP\textsubscript{2} (Table 6.4).

Figure C.17: AtomicContract AC\textsubscript{42} that is used to express PP\textsubscript{3} (Table 6.4).
C.3. FORMULATION OF PATTERN CONTRACTS

Figure C.18: AtomicContract AC43 that is used to express PP4 (Table 6.4).

Figure C.19: AtomicContract AC44 that is used to express PP5 (Table 6.4).
Appendix D

UML-RT-to-Kiltera: Source and Target Metamodels

In this appendix, we demonstrate the complete source UML-RT metamodel and the target Kiltera metamodel of the UML-RT-to-Kiltera model transformation.
Figure D.1: The source UML-RT metamodel of the UML-RT-to-Kiltera model transformation.
Figure D.2: The target Kiltera metamodel of the UML-KT-to-Kiltera model transformation.