Union Models: Support of Variability Modeling and Efficient Reasoning about Model Families over Space and Time

Sana'a Abdel Ra'uof Alwidian

Thesis submitted in partial fulfillment of the requirements for the

Doctorate in Philosophy degree in Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

uOttawa

University of Ottawa

Ottawa, Ontario, Canada

February 2020

© Sana'a Abdel Ra'uof Alwidian, Ottawa, Canada, 2020

Abstract

In Model-Driven Engineering (MDE), models change continuously due, for instance, to new requirements or refined understanding about the domain, resulting in what we call *model families*. For a given modeling language, a *model family* is a set of related models, with commonalities and variabilities between individual models. Model families stem from the evolution of models into several *versions* over *time* and/or the *variation* of models over the *space* dimension. In contexts where there are several versions/variations of a model, analyzing individual models, *one model at a time*, becomes cumbersome and inefficient, especially when many models share several elements in common (i.e., redundancy). In addition, other kinds of analyses that rely on temporal or spatial information (e.g., trend analysis over time) become complex using individual models, separately.

This thesis proposes *union models* as first-class generic artifacts to: 1) support the representation of model families (for time and space dimensions) using one generic model; 2) achieve performance gains during analysis of family models *all at once*, compared to the analysis of individual models, *one model at a time*; and 3) support types of analyses that are more easily feasible with union models compared with individual models.

The use of union models is challenging and non-trivial. At the *model level*, the challenges stem mainly from the following requirements: (Req.1) Models in a family shall be captured (in both dimensions of variability) in a *complete* and *exact* way such that all and only individual members of a family are included in one model. (Req.2) The resulting union model shall be as *compact* as possible, in the sense that it should not contain redundant elements, especially when there are many elements in common between models. (Req.3) The union model shall be *self-explanatory*, i.e., it should be supported with a mechanism that distinguishes which elements belong to which models. There are also other challenges at the *metamodel level* associated with the use of union models. In particular, a union model may *not* be a valid instance of the language's metamodel, and the latter might need to have its constraints relaxed accordingly.

We demonstrate, empirically, the usefulness of union models for analyzing a family of models, *all at once*, compared to individual models, *one model at a time*. The **contributions** of the thesis are:

Major: (1) a *language-independent*, graph-based formalization of model families and union models, (2) a generic, language-independent *algorithm* to produce a union model from a set of models (in a compact and exact manner) in a given language (to satisfy Req1. and Req.2), (3) a *spatio-temporal annotation language* (STAL) to support the representation of variability in model families in the space and time dimensions, and to facilitate reasoning about union models (to satisfy Req.3), (4) improved *efficiency* of analysis and reasoning over a set of models, all at once using union models, compared to reasoning on single models, one model at a time, and (5) *examples* of analysis techniques adapted to support efficient reasoning about model families.

Minor: this thesis also addresses the metamodel-level challenges associated with union models. In particular, it contributes a characterization of the requirements for minimally relaxing modeling languages to support all potential union models of a language. The thesis proposes two methods (Evi-MeReFam and Anti-MeReFam) to *infer/anticipate* relaxation points, (i.e., locations where metamodel relaxations are needed) so that existing tools and analysis techniques would be adapted *once* per language.

Acknowledgment

First of all, I thank ALLAH, the almighty, for helping me and giving me the strength and patience to complete this work.

I would like to express my sincere gratitude, appreciation and respect to my supervisor, Professor Daniel Amyot, for his continuous support of my Ph.D. study and related research. His inspirational thoughts, valuable guidance and assistance, patience and careful review helped me in all the time of research and writing of this thesis. He provided me with valuable support and gave generously his time, wisdom and experience. I could not have imagined having a better advisor and mentor during my journey in the Ph.D.

My deepest gratitude and appreciation go to the province of Ontario for granting me their prestigious and generous scholarship, the *Ontario Trillium Scholarship (OTS)*. Without this scholarship, pursuing my dream of getting a Ph.D. would have been impossible. My special thanks are also dedicated to the University of Ottawa, for adding another dimension to my horizon, and for granting me their generous scholarships: the *University of Ottawa Excellence Scholarship* and the *International Doctoral Scholarship*. My sincere gratitude goes also to the *BMO Financial Group Graduate Bursaries* for granting me their generous scholarship, to *NSERC* (Discovery program) and the *Ontario Graduate Scholarship (OGS)* for their funding in the last semesters of my Ph.D.

I would also like to thank my Ph.D. committee members, Prof. Dorina Petriu, Prof. Stéphane Somé, Prof. John Mylopoulos, and Prof. Nan Niu for their insightful comments and feedback, which incited me to view my research from various perspectives. Their support and help highly contributed to the success of this work. I am also deeply grateful to people who gave me feedback and commented on my research, including Michalis Famelis, Houari Sahraoui, Rick Salay, Alicia Grubb, Omar Allam, Daniel Strüber, Sahar Kokaly, Nelly Bencomo, Alfonso Pierantonio, and Yngve Lamo. In addition, I am very thankful to my colleague Amal A. Anda for her help with CPLEX.

My heartfelt thankfulness goes to my lovely and always caring mother, Yasar, and father, Abdel Raouf, who have always encouraged me towards success, tolerated me during

my studies and always directed me to take the right way. Without their encouragement and prayers, this work would never exist. My sincere thanks to my sisters: Somayah, Suha, Samah and Sakhaa and brothers: Mohammad, Motaz and Ahmad. Without your support and having you around, I would not have been able to have reached this far.

Furthermore, I deeply appreciate the encouragement and support shown by my friends, cousins, aunts, and my parents in law, Narjes and Naser. I could not finish this part without also thanking my sisters in law: Areej, Alaa (whose name in English has the same spelling as my husband's name!), Ayat, Asma, Shaima and Baraah.

Marie, my little big daughter, thank you for making my life extraordinary! Writing my thesis while, *at the very same time*, singing to you "Twinkle twinkle little star…" taught me the real meaning of efficient multitasking! You are my Ph.D. baby and my greatest contribution ever. All of this would never been enjoyed without the action that you have added to my Ph.D. journey. You made it more challenging and rewarding.

Most of all, I stand speechless when it comes to thank my friend, life partner, Ph.D. colleague, office neighbour at Ottawa U, husband and companion, love and life, Alaa. Trying to thank you with all possible languages in the earth is still not enough! Thank you for your unconditional endless love. Thank you for your care, patience, tolerance and noble manners. Thank you for your encouragement and wisdom. Thank you for believing in me, and making me believe in myself in a different way, especially under hard circumstances. Being two Ph.D. students in the same house is not trivial at all! Thank you for your "not-so-well" hidden effort in balancing between your Ph.D. work and mine, and for prioritizing my work over yours most of the time. Thanks for making sure that I am surrounded with everything I need to focus and progress. This would not have been possible without you. Now it is my turn to nurture your dreams and encourage you towards pursuing your Ph.D.

Dedication

-To my husband Alaa and my daughter Marie, without whom this thesis would have been completed two years earlier… but also without whom my life would be empty and meaningless!

-To the soul of my father, who fought for me to be here!

Table of Contents

Abstract ii

Acknowledgment iv

Dedication vi

Table of Contents vii

List of Figures xi

List of Tables xiv

List of Acronyms xv

Glossary xvii

Chapter 1. Introduction 1

1.1. Problem Specification 2

1.2. Motivation 3

1.3. Other Challenges at the Metamodel Level 7

1.4. Thesis Objectives and Scope 9

1.5. Research Questions 11

1.6. Methodology 12

1.7. Thesis Contributions 13

1.8. Publications 13

1.9. Thesis Outline 15

Chapter 2. Literature Review 16

2.1. Review Methodology 16

2.2. Research Area 1: Variability Modeling in MDE in Space or Time 18

2.2.1 Overview 18
2.2.2 Model Driven Engineering 18
2.2.3 Software Product Line Engineering (Variability in Space) 20
2.2.4 Version Control (Variability in Time) 21

2.3. Model-Driven Product Line Engineering (MDE+SPLE) 22

2.3.1 Modeling Variability in SPLE 22
2.3.2 Variability Modeling Approaches 24

2.4. Model Version Control (MDE+VC) ... 29

2.5. Software Product Line Version Control (SPLE + VC) ... 33

2.6. Models as SPL in Space or Time ... 35

2.7. Variability Management Using Union Models-like Artifacts ... 37

2.8. Variability Modeling using Goal Models ... 39

2.9. Research Area 2: Product Line Analysis ... 41
2.9.1 Product-based Analysis ... 41
2.9.2 Family-based Product Line Analysis ... 42
2.9.3 Feature-based Analysis ... 45

2.10. Research Area 3: Metamodel-Related Evolution Approaches ... 46
2.10.1 Overview ... 46
2.10.2 Metamodel Evolution and Model Co-evolution ... 47
2.10.3 Metamodel Evolution and Transformation Co-evolution ... 49
2.10.4 Metamodel Evolution and OCL Co-evolution ... 50

2.11. Metamodel Extension Approaches ... 51

2.12. (Meta)Model Versioning Approaches ... 51

2.13. Metamodel Relaxation Approaches ... 52

2.14. Uncertainty Management Approaches ... 54

2.15. Database Schema Evolution Approaches ... 55

2.16. Chapter Summary ... 56

Chapter 3. Characterization of Model Families ... 59

3.1. Model Families ... 59
3.1.1 Preliminary: Software Families ... 59
3.1.2 Variability in Space and Time ... 60
3.1.3 Model Families: An Analogy with Software Families ... 62

3.2. Special Characteristics of Model Families ... 65

3.3. Challenges of Model Families ... 66
3.3.1 Variability-related Challenges at the Model Level ... 66
3.3.2 The Need for Generic Models to Represent Model Families ... 67
3.3.3 Conformance-related Challenges at the Metamodel Level ... 68

3.4. Spatio-Temporal Annotation Language (STAL) ... 69
3.4.1 Syntax and Semantics of STAL ... 69
3.4.2 Examples ... 71
3.4.3 Merging of STAL Expressions ... 73

3.5. Chapter Summary ... 78

Chapter 4. Formalization of Model Families ... 80

4.1. Why Formalization ... 80

4.2. Graph-based Formalization of (Meta) Models. ... 81

4.2.1 Graphs and Graph Morphisms 81
4.2.2 Type(d) Graphs and Typed Graph Morphisms 82
4.2.3 Attributed Type(d) Graphs (as E-Graphs) 83
4.2.4 Constraints 86

4.3. (Meta)models as Attributed Type(d) Graphs with Constraints 88

4.4. Formalization of Model Families and Union Models 88
4.4.1 Formalization of Model Families 88
4.4.2 Formalization of Union Models 90

4.5. Propositional Encoding Language with Annotation (PELA) 91

4.6. Union of Propositional Encodings of Models 95
4.6.1 Example 95
4.6.2 Union of M_U and Another Model 99
4.6.3 Decisions while Constructing M_U: 101

4.7. Chapter Summary 102

Chapter 5. Analysis and Reasoning with Model Families 103

5.1. Reasoning Tasks 103
5.1.1 RT1: Property Checking 103
5.1.2 RT2: Trend Analysis 106
5.1.3 RT3: Commonality Analysis 106

5.2. Experiments 106
5.2.1 Methodology 107
5.2.2 Implementation 109

5.3. Results for Property Checking (RT1) 110

5.4. Results for Trend Analysis (RT2) 113

5.5. Results for Commonality Analysis (RT3) 115

5.6. Threats to Validity 117

5.7. Chapter Summary 117

Chapter 6. Adapting Analysis Techniques for Goal Model Families 119

6.1. Goal-oriented Requirement Language 120
6.1.1 GRL Syntax 120
6.1.2 Propagation-based Analysis in GRL 120

6.2. Forward Propagation Analysis in Standard GRL 121
6.2.1 Arithmetic Semantics of GRL Forward Propagation Algorithm 121
6.2.2 Adapted Semantics for GRL Model Families 125

6.3. Backward Propagation Analysis in Standard GRL 126
6.3.1 Semantics of GRL Backward Propagation Algorithm 127
6.3.2 Adapted Semantics for GRL Model Families 129

6.4. Illustrative Case Study 132

6.5. Experiments 134

6.5.1 Methodology 135
6.5.2 Results for SIZE=S and INDV=(S, M, L, XL) 138
6.5.3 Results for SIZE=M and INDV=(S, M, L,XL) 141
6.5.4 Results for SIZE=L and INDV=(S, M, L,XL) 144
6.5.5 Results for SIZE=XL and INDV=(S, M, L, XL) 147
6.5.6 Summary of All Results 150

6.6. Chapter Summary 151

Chapter 7. Metamodel Relaxation to Support Model Families 153

7.1. Why Metamodel Relaxation? 154
7.1.1 Scenario One: Different GRL Actors with the Same Goal 154
7.1.2 Scenario Two: GRL Intentional Elements with Multiple Links 155
7.1.3 Scenario Three: UML Attributes with Multiple Types 157

7.2. Proposed Solution 1: Evidence-Based Relaxations 157
7.2.1 Phase One: Construct the Union Model 158
7.2.2 Phase Two: Detect Non-conformance 159
7.2.3 Phase Three: Infer Metamodel Relaxation Points 160
7.2.4 Illustrative Example of Evi-MeReFam 161
7.2.5 Limitations of Evidence-based Relaxation 164

7.3. Solution 2: Anticipation-Based Relaxation 165
7.3.1 Anticipation Based on Structural Similarity in the Same Metamodel 165
7.3.2 Anticipation Based on Structural Patterns across Different Metamodels 167
7.3.3 Discussion 172

7.4. Chapter Summary 173

Chapter 8. Conclusions and Future Work 175

8.1. Summary and Contributions 175

8.2. Differences between this Thesis and Closely-Related Work 176

8.3. Threats to Validity 178
8.3.1 Construct Validity 178
8.3.2 Internal Validity 179
8.3.3 External Validity 179

8.4. Future Work 180

References 183

Appendix A: GRL Metamodel and Syntax 200

Appendix B: Summary of STAL's Syntax 203

Appendix C: Sample CPLEX Code 207

List of Figures

Figure 1 Overview of background concepts and their relationships.................... xviii
Figure 2 Goal model family of regulations.. 4
Figure 3 Goal model family of regulations in space and time 5
Figure 4 A union model for the model family in Figure 2....................................... 7
Figure 5 Example (using the GRL language) of a union model that does not conform to the metamodel of the individual models in the model family... 8
Figure 6 Model family-specific metamodel conformance problem.......................... 9
Figure 7 Scope of the thesis 11
Figure 8 Organization of literature review for research area 1 19
Figure 9 Modeling hierarchy in MDE [19].. 19
Figure 10 (a): Negative variability, (b): positive variability and (c) delta modeling (adapted from [48])... 28
Figure 11 Types of merging (adapted from [59]) 31
Figure 12 State-based and Operation-based model merging (from [59]) 32
Figure 13 Metamodel (MM) evolution and model (M) co-evolution problem......... 47
Figure 14 Model-triggered metamodel (MM) evolution problem (general)............ 47
Figure 15 Microsoft Office 2016 as a software family with variability in space (i.e., configurations)... 61
Figure 16 MS Word as a software family with variability in time (i.e., evolution).. 61
Figure 17 Microsoft Office software family with different configurations (vertical) and several versions (horizontal) [73] 62
Figure 18 (a) MvF with several versions over time, (b) McF with several configurations 64
Figure 19 Model families in both space and time dimensions................................. 64
Figure 20 A more realistic model families in both space and time dimensions 65
Figure 21 Homogeneous models (a model family).. 66
Figure 22 Heterogeneous models (out of thesis scope) 66
Figure 23 Goal model family of regulations (from Figure 2) 69
Figure 24 Annotated union model of the model family in Figure 23 72
Figure 25 Annotated union model of the model family in Figure 3 73
Figure 26 Binary relation for expression Exp1 74
Figure 27 Binary relation for expression Exp2 75
Figure 28 Binary relation for expression (Matrix)Result... 78
Figure 29 Relationship between models and their graph representation 81
Figure 30 Graph morphism between graphs 82
Figure 31 (a) Type graph, and (b) Typed graph... 83
Figure 32 Typed graph morphism.. 83
Figure 33 A typed attributed graph typed over the ATG in Figure 34 83
Figure 34 An attributed type graph (ATG) 84
Figure 35 A visualization of an E-graph.. 85
Figure 36 An E-graph (EG) with node attributes represented as edges.................... 85

Figure 37 An instance typed attributed graph of the E-graph in Figure 36 86
Figure 38 Model family morphism 89
Figure 39 Model family morphism as a composition of typed graph morphisms 89
Figure 40 *GraphToPropositionwithAnnotation* encoding of model M in Figure 37 94
Figure 41 First version of a state transition diagram, M1 96
Figure 42 Representation of M1 as an E-graph 96
Figure 43 Propositional encoding of M1 96
Figure 44 Second version of a state transition diagram, M2 96
Figure 45 Representation of M2 as an E-graph 97
Figure 46 Propositional encoding of M2 97
Figure 47 Union of the propositional encodings of M1 and M2 98
Figure 48 Representation of MU as an E-graph 98
Figure 49 Conventional representation of MU as an annotated state transition diagram 99
Figure 50 Third version of a state transition diagram, M3 99
Figure 51 Representation of M3 as an E-graph 99
Figure 52 Propositional encoding of M3 100
Figure 53 Union of the propositional encodings of PEU and M3 100
Figure 54 Conventional representation of MUnew as an annotated state transition diagram 101
Figure 55 An example of propositional encoding of a property 104
Figure 56 (a) An acyclic model M1, (b) an acyclic model M2, (c) a union model of M1 and M2 that is not acyclic 105
Figure 57 Summary of results for property checking (RT1), for all *INDV* and *SIZE* categories 112
Figure 58 Summary of results for trend analysis (RT2), for all INDV and SIZE categories 114
Figure 59 Summary of results for commonality analysis (RT3), for all *INDV* and *SIZE* categories 116
Figure 60 OR-type decomposition links and their arithmetic semantics 122
Figure 61 AND-type decomposition links and their arithmetic semantics 122
Figure 62 Contribution links and their arithmetic semantics 123
Figure 63 Dependency links and their arithmetic semantics 123
Figure 64 Multiple types of links and the arithmetic semantics of their propagation 124
Figure 65 An actor containing multiple IEs with their importance values, and its arithmetic semantics 124
Figure 66 A GRL model with multiple actors, and the arithmetic semantics of M's quantitative evaluation 125
Figure 67 Examples of three model versions (a)-(c), and their union model MU (d) 126
Figure 68 A solution for MU when G5=25 131
Figure 69 A solution for MU when G5=25 and G2=50 131
Figure 70 Goal model family for smart home environments varying according to space and time 133

Figure 71 A comparison between T_{ind} and T_{MU} for small-sized models (S), with different numbers of models/families 139
Figure 72 Speedup achieved by an M_U of small-sized model (S).......................... 139
Figure 73 Time saving achieved by an M_U of small-sized model (S).................... 140
Figure 74 Additional memory consumed by an M_U of small-sized models (S), as a percentage of the memory consumed by the largest individual model ... 141
Figure 75 A comparison between T_{ind} and T_{MU} for medium-sized models (M), with different numbers of models/family 142
Figure 76 Speedup achieved by an M_U of medium-sized models (M) 143
Figure 77 Time saving achieved by an M_U of medium-sized models (M) 143
Figure 78 Additional memory consumed by an M_U of medium-sized models (M), as a percentage of the memory consumed by the largest individual model. 144
Figure 79 A comparison between T_{ind} and T_{MU} for large-sized models (L), with different numbers of models/family 145
Figure 80 Speedup achieved by an M_U of large-sized models (L) 145
Figure 81 Time saving achieved by an M_U of large-sized models (L).................. 146
Figure 82 Additional memory consumed by an M_U of large-sized models (L), as a percentage of the memory consumed by the largest individual model ... 147
Figure 83 A comparison between T_{ind} and TM_U for XL-sized models, with different numbers of models/family 148
Figure 84 Speedup achieved by an M_U of XL-sized models 148
Figure 85 Time saving achieved by an M_U of XL-sized models 149
Figure 86 Additional memory consumed by an M_U of extra-large models (XL), as a percentage of the memory consumed by the largest individual model ... 150
Figure 87 A summary of T_{ind} for all SIZE categories 151
Figure 88 First GRL example: SomeGoal moves from ActorA to ActorB with their union model and relevant GRL metamodel elements............................. 154
Figure 89 Extract of the GRL metamodel 155
Figure 90 (a) Version 1: GoalA and Task2 with decomposition link (b) Version 2: GoalA and Task2 with Contribution Link (c) union model that regroups both models... 156
Figure 91 Two UML classes and their resulting union model............................... 157
Figure 92 A Conceptual Architecture of the Evi-MeReFam Method.................... 158
Figure 93 The set of models in MF (M_0, M_1, and M_2) and their union model M_U. 158
Figure 94 Characterization of GRL's metamodel (internal) constraints................ 163
Figure 95 Examples where merging elements differently affects MM violations. 164
Figure 96 Example of structural similarity between classes in GRL's metamodel. 166
Figure 97 (a) Basic metamodel's components design pattern, and (b) containment design pattern, adapted from [226]. 169
Figure 98 A simplified GRL metamodel tagged with general concepts................ 170
Figure 99 Structural patterns in GRL metamodel. 171
Figure 100 Structural patterns in (a): STD metamodel, and (b): FM metamodel.... 172
Figure 101 Extract of the GRL metamodel (ITU-T [6]) 200
Figure 102 Summary of the GRL graphical syntax 201

List of Tables

Table 1 Mapping configurations to their detailed descriptions 70
Table 2 Categories of parameter *SIZE* (number of elements in a model)............. 108
Table 3 Categories of parameter *INDV* (number of individual members in a family) 108
Table 4 Categories of parameter *SIZE* (number of elements in a model)............. 135
Table 5 Categories of parameter *INDV* (number of models in a family) 136
Table 6 Memory consumption by an M_U of small-sized model 140
Table 7 Memory consumption by an M_U of medium-sized model....................... 143
Table 8 Memory consumption by an M_U of Large-sized model........................... 146
Table 9 Memory consumption by an M_U of XL-sized model.............................. 149
Table 10 A constraint relaxation map (ConReMap).. 166
Table 11 Constraint similarity map (ConSiMap) between MM constraints........... 166
Table 12 Anticipated relaxation points based on constraint similarity.................. 167

List of Acronyms

Acronym	Definition
AMOR	Adaptable Model Versioning
Anti-MeReFam	*Anti*cipation-based *Me*tamodel *Re*laxation for model *Fam*ilies
AoURN	Aspect-oriented User Requirements Notation
ATG	Attributed Type Graph
ATL	Atlas Transformation Language
CP	Constraint Programming
CVL	Common Variability Language
DSML	Domain-Specific Modeling Language
EMF	Eclipse Modeling Framework
Evi-MeReFam	*Evi*dence-based *Me*tamodel *Re*laxation for model *Fam*ilies
FM	Feature Model
G_i	Typed Graph *i*
G_U	Graph Union
GRL	Goal-oriented Requirement Language
HFM	Hyper Feature Models
IPC	Implicit Presence Condition
ITU	International Telecommunication Union
ITU-T	ITU-Telecommunication Standardization Sector
M	Model
M_U	Union Model
MBE	Model-Based Engineering
McF	Model-configurations Family
MCV	Model Version Control
MDE	Model-Driven Engineering
MDPLE	Model-Driven Product Line Engineering
ME	Meta Expression
MF	Model Family
MM	Metamodel
MM_U	Metamodel that supports union models
MOF	Meta Object Facility
MoSo-PoLiTe	Model-based Software Product Line Testing
MoVaC	Model Variant Comparison
MTTL	Model Template Transformation Language
MVDM	Multi-Variant Domain Model
MvF	Model-Versions Family
NFR	Non-Functional Requirements
NSGAII	Non-dominated Sorting Genetic Algorithm II
OCL	Object Constraint Language
OPL	Optimization Programming Language

OVM	Orthogonal Variability Modelling
P2PP	Peer-to-Peer Protocol
PC	Presence Condition
PDF	Portable Document Format
PELA	Propositional Encoding Language with Annotation
PLiBS	Product Line Behavioral Synthesis
PLUS	Product Line UML based Software Engineering
QVT	Query/View/Transformation
SAT	Boolean satisfiability problem
SECO	Software Ecosystem
SPL	Software Product Line
SPLE	Software Product Line Engineering
SPLVC	Software Product Line Version Control
STAL	Spatio-Temporal Annotation Language
TG	Type Graph
UCM	Use Case Maps
UML	Unified Modeling Language
URN	User Requirements Notation
UUID	Universal Unique Identifier
VC	Version Control

Glossary

This glossary introduces the necessary concepts (and their relationships) required to understand this research. In particular, definitions are inferred based on the literature for the following concepts: model, metamodel, feature model, as well as the conformance relationship between models and metamodels. In addition, this glossary defines and specifies the meaning of variability/variation, model family, and union model used in the context of this thesis. The formalization of some of these concepts is elaborated further in Chapter 4.

- **Model (M):** A model is an abstract representation of entities, properties, and relationships of a particular system. One important requirement of a model is to conform to a metamodel.
- **Metamodel (MM):** A metamodel is a model that describes the abstract syntax and the static semantics (such as multiplicities of elements involved in relationships) of a modeling language [1]. In the Model-Driven Engineering (MDE) community, metamodels are often specified with Unified Modeling Language (UML) class diagrams complemented by Object Constraint Language (OCL) constraints [2].
- **Conformance**: A model is said to *conformTo* (or instantiate) its metamodel if the former obeys the rules and constraints imposed by the metamodel.
- **Variability**: The concept of variability is often used in software product lines (SPL) [3], but this work expands it to arbitrary modeling languages. Variability in this thesis means "*the existence of multiple versions or variations of an initial model*". This thesis considers the following reasons for model variability: (1) model *evolution*[1] over time, which produces model *versions*, and (2) model *variation* over the space dimension, which produces multiple *variations* or *configurations*[2].
- **Model Family (MF):** For a given modeling language, a model family is a set of related models, with similarities and differences between individual models, that results from

[1] Throughout this thesis, the word "evolution" is used to express changes over the time dimension, and the word "variation" is used to express changes over the space dimension.
[2] This thesis uses the words "variations" and "configurations" interchangeably.

the variation/evolution of models over the space and/or time dimensions. Each model in a model family conforms to the same metamodel since these models are expressed using the same language. A model can be part of multiple families.

- **Feature Model (FM):** The variabilities and commonalities (and their dependencies/constraints) of a model family can be represented as features in a feature model. A member of a model family can be produced or derived through the use of a valid feature configuration from the feature model [4].
- **Union model (M_U):** A model that represents the union of all elements in all individual models of a model family, captured in a compact, exact, complete and self-explanatory manner, to facilitate an efficient analysis of a group of models, all at once. Since a union model captures all member models that conform to a specific metamodel, their union should also ideally conform to the same metamodel, but this is typically not guaranteed (as discussed in Chapter 7).

Figure illustrates the relationships between these concepts in the form of a UML class diagram.

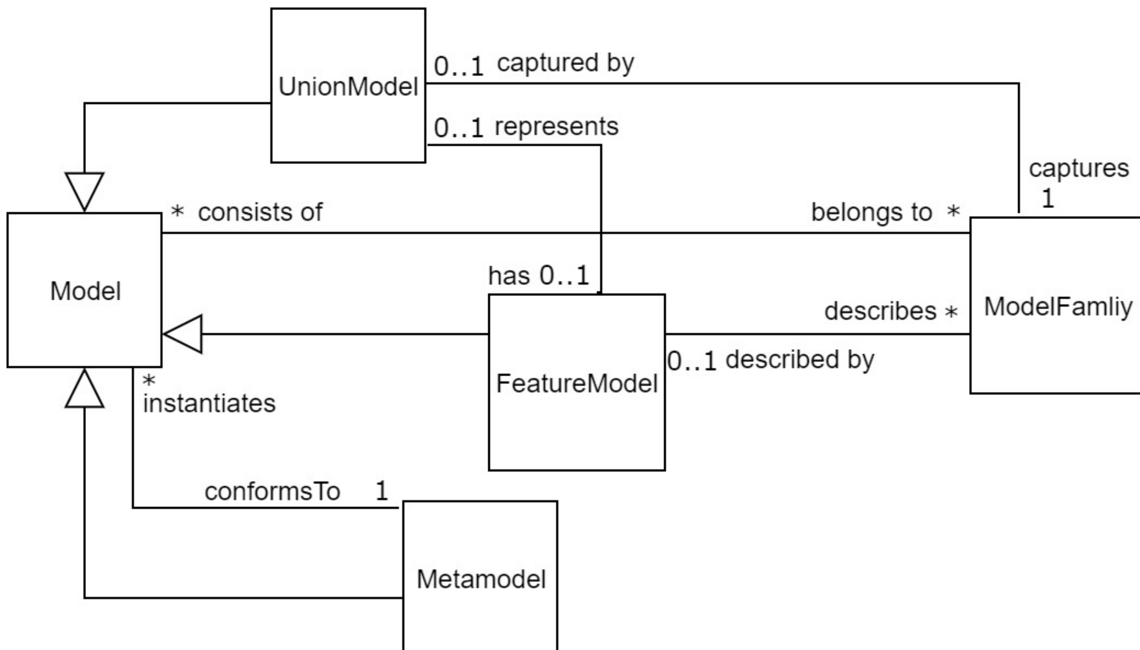


Figure 1 Overview of background concepts and their relationships

Chapter 1. Introduction

This thesis proposes a new modeling paradigm to model and reason about *model families* that change along the *time* and/or *space* (or product) dimensions. This paradigm is realized through the use of *union models* (M_U) as first-class, generic artifacts with the following characteristics:

1) An M_U captures family members (i.e., models that evolve) in a *complete* and *exact* way, such that all and only individual members of a given family are captured;
2) An M_U provides *compact* representation of a model family. That is, M_U does not contain redundant elements, which is useful when there are many elements in common between individual models; and
3) An M_U captures the multiple versions/variations of family members in both the space and time dimensions in an *explicit* and *self-explanatory* manner. To realize this feature, the thesis contributes a *Spatio-Temporal Annotation Language* (STAL) to annotate model elements with information about space and time and to distinguish which elements belong to which models.

The major goals behind union models are not only to represent the entirety of a model family using one generic model, but also to facilitate a more efficient analysis over a group of models, *all at once*, compared to the analysis of individual models, *one model* at a time, and also to support types of analysis that are easier with union models than with individual models. This thesis categorizes a set of analysis techniques that are deemed beneficial when used on union models and illustrates their enhanced performance in comparison with classical analysis on individual models.

To support the representation of union models as valid instances of a language at the *metamodel* level, this thesis also suggests two methods, named *EviMeReFam* and the *AntiMeReFam*, to infer/anticipate metamodel relaxations points. The proposed methods target the relaxation of metamodel constraints related to multiplicities of association ends and attribute types and to the external constraints of languages (e.g., expressed in OCL).

This chapter discusses the problem specification, the motivation, as well as the objectives and scope of this research. Furthermore, the research questions, methodology, and lists of existing contributions and publications based on this research are discussed. Finally, the organization of this thesis is outlined.

1.1. Problem Specification

In Model-Based Engineering (MBE), models are the first-class artifacts used to represent and abstract knowledge and activities that govern a particular domain [5]. In MBE, models in any given language often undergo continuous changes during their lifecycle, often due to modifications of requirements or standards, or to enhanced understanding of the domain to be modeled. Such change could happen over the course of *time* (i.e., *evolution*), resulting in one model evolving into a set of related *versions*. A model could also vary over the *space* dimension, where there could be several *variations* (or *configurations*) of the same model, all existing at the same time. Such variations are commonly used to represent slightly different products. In both scenarios, a *family* of related models in the same language, with commonalities and variabilities between family members, is called a *model family*.

Change in an MBE context is inevitable. Hence, raising awareness to the phenomena of model families is of particular importance, especially in variant-rich domains such as cyber-physical systems, smart systems, or regulatory environments (where slightly different regulations need to be modeled for different regulated parties and jurisdictions). In any of these domains, models that are used to capture the domain's dynamic nature are subject to frequent variations and evolutions. In other words, a modeler may start with an initial model version (v_0), which over time needs to be updated into a slightly different version (v_1) to reflect a changing requirement. This version may further evolve into versions v_2, v_3, and so on. In the space dimension, two or more modelers may need, at the same time, to create slightly different variations of an initial model to reflect different products or *configurations*. In such contexts, modelers often end up having a family of model versions and/or variations.

Often, there is a need to analyze the elements of a model family to check some property or compute some trend. Yet, analysis of, and reasoning about each of these family members, *individually*, can become inefficient and time consuming. This is because having

a group of models to be analyzed requires the modeler to load into a tool, analyze, and report on analysis results of each individual model, separately. This is a time-consuming and laborious process, which becomes even more critical as the number of models to analyze gets larger. More importantly, analyzing a set of related models, with typical similarities, often involves redundant computations and may even require repeated user assistance (e.g., for interactive analysis). This inefficiency is especially a problem if there is a large set of models with high proportion of common elements (i.e., redundancy) across models. In this case, analyzing members of a model family, *all at once*, is more desirable than analyzing each individual model, *one at a time*, for efficiency reasons.

The above-mentioned challenges motivate the need to find a way to analyze model families, all at once, other than analyzing individual models, one model at a time. The alternative solution should provide the ability to represent or capture the entirety of all model family members in *one generic artifact*, in such a way that individual models are captured *compactly* (i.e., without redundancy), *exactly* (i.e., all and only family members) and in an *explicit* and a *self-explanatory* manner. That is, elements of different family members should be mapped or annotated with information about their space and time context, so as to distinguish which element belongs to which model. The generic modeling artifact should also enable a faster and a more efficient analysis of the time/space-evolving family models, *all at once*, compared to analyzing one individual model at a time. Furthermore, the alternative solution should facilitate types of advanced analysis that are either infeasible or inefficient with individual models.

1.2. Motivation

This work is inspired by issues faced previously with regulation modeling, in collaboration with Transport Canada, where there are regulations that evolve over time and that apply to different types of organizations (i.e., spaces). We explain the challenges associated with regulatory model families, and motivate our proposed solution, using a running example from this domain, namely airports regulated by Transport Canada, where we use the Goal-oriented Requirement Language (GRL) as a modeling language. GRL is a part of the User Requirements Notation (URN) standard [6][7], and its syntax is summarized in Appendix A.

Without loss of generality, the subsequent discussion about GRL model families in regulatory domains and their challenges is also applicable to other model families from modeling languages other than the GRL. Therefore, we posit that our approach is feasible for any metamodel-based modeling languages and their model families.

In Transport Canada, there are many regulations that need to be modeled by regulators, for example, to enable compliance and performance assessments. These regulations evolve over time and apply to multiple types of organizations (such as aerodromes and airlines) that are also of different sizes [8][9]. For example, as Figure 2 shows, some rules/regulations apply differently depending on the targeted aerodrome type (modeled abstractly here as TYPE1, TYPE2, and TYPE3), where some regulations are only applicable to specific aerodrome types. In Figure 2, regulations related to *Fence rule 3* are only applicable to aerodrome TYPE1, Access Control System (ACS) *rule 2* is applicable to all aerodrome types, and *ACS rule 1* is applicable to TYPE1 and TYPE2 aerodromes but not to TYPE3 ones.

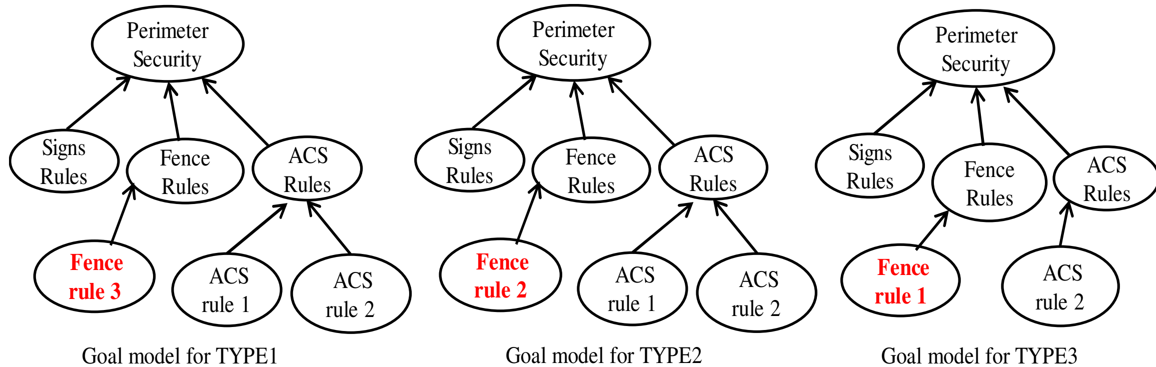


Figure 2 Goal model family of regulations

The different aerodrome types (i.e., TYPE1, TYPE2 and TYPE3) can be considered as different *configurations* (or *spaces*) that are available at the same time for different aerodromes. This means that the three goal model variations depicted in Figure 2 represent a goal model family along the space dimension, where each space has different regulations. Goal models in this family could also evolve over *time* (e.g., when the regulatory context evolves) resulting in several *versions* of the same model. Evolution of goal models could involve the addition/deletion of goals and/or links, or modifications to attributes of goals

and/or links, as illustrated in each row of Figure 3. In this figure, the (asynchronous) changes across each of the initial configuration models (from Figure 2) are highlighted.

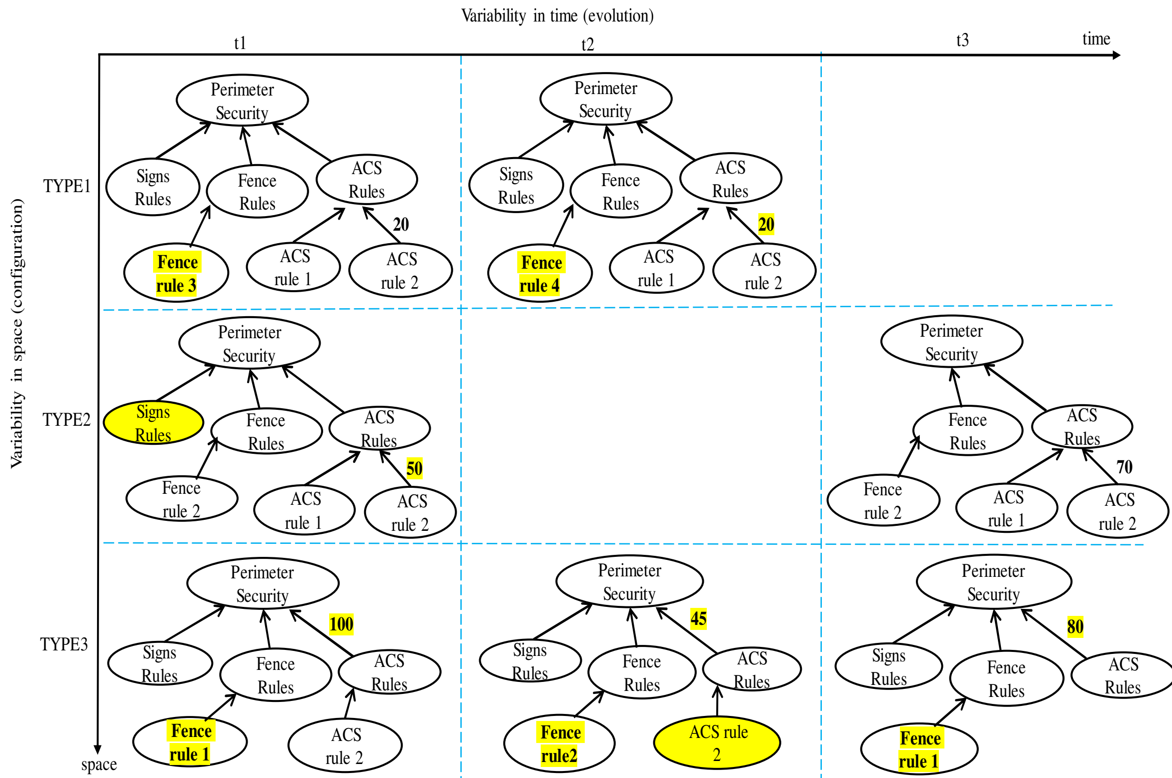


Figure 3 Goal model family of regulations in space and time

Analyzing the different versions/variations of models in Figure 3 *individually* using one goal model per type of aerodrome (and at each time instance) is *impractical* for the following reasons:

1) If a modeler plans to conduct goal satisfaction analysis (using some GRL forward propagation algorithm [6]) on each individual model by running a strategy that initially assigns "*ACS rule 2*" to study the impact of its satisfaction on the satisfaction of other goals, she would end up running the same evaluation algorithm seven times (for this example only), even though there are several common elements among the seven models. Intuitively, if there are M individual models in a model family, and each model has E elements, then the complexity of running a satisfaction propagation algorithm on all models would be in order of $O(M \times E)$. Such complexity becomes more significant if there are hundreds of models, with hundreds of elements in each model, which is not an atypical situation.

2) Each individual model per se (e.g., goal model for *TYPE1* at time *t1*) does not represent the whole set of regulations. Hence, a regulator who wishes to reason about all regulations (e.g., to study the evolution trend of regulations over time) would have to check all models, *one model at a time*, and reason about each one separately. This process becomes inefficient and time consuming as the number of models increases.
3) Models in a model family are subject to frequent evolutions over time that are *asynchronous* by nature. Unsynchronized evolutions sometimes require that older versions of models (which represent *legacy* models) need to be maintained as they may still be in use even after they were superseded by newer versions. This is an issue over the time dimension which can also be witnessed in the space dimension (i.e., along configurations). In this scenario, legacy models as well as new models need to co-exist together in one model to be analyzed together.

Similar challenges exist in the software engineering field, particularly in the Software Product Line (SPL) domain, where products vary over the space dimension, and may also evolve over time. Since, by nature, an SPL encodes a set of related product variants, then dealing with the evolution of multiple products over time means that developers should consider an additional dimension of variability, and hence, reason about sets of products. Such combination of variation and evolution is not well supported by existing SPL engineering techniques, and only a few recent approaches in the literature try to address this issue. For example, Famelis et al. [10] proposed an approach for combining SPLs with partial models to represent design-time uncertainty. The resulting artifact, called Software Product Lines with Design Choices (SPLDCs), integrates variability and design uncertainty in a common formalism to help developers differentiate between the kinds of decisions that are relevant during the design and configuration stages of the SPL lifecycle. Seidl et al. [11] and Lity et al. [12] also considered variation of SPLs in space and time and proposed a so-called *175% modeling* formalism to allow for the development and documentation of evolving product lines. They also urge the community to extend annotative variability modeling to tackle evolution and variation by the same means.

The above-mentioned issues (especially the first one), and the gaps identified in the literature, motivate the need to find a way of representing model families other than using

separate individual models. This thesis proposes *union models* (M_U) as a single generic model to capture the entirety of the space/time-evolving model families (in both dimensions of variability), in a comprehensive and exact way such that analysis of a group of models will be faster than for individual models, and where members of a family can be extracted and analyzed. For illustration purposes, an M_U that captures the model family shown in Figure 2 is represented in Figure 4, where all elements (i.e., goal and links) of the union model are annotated with space information (TYPES in this example) to distinguish which element belongs to with model variant. More details about representing union models and their associated annotation language are presented in Chapter 3 and Chapter 4.

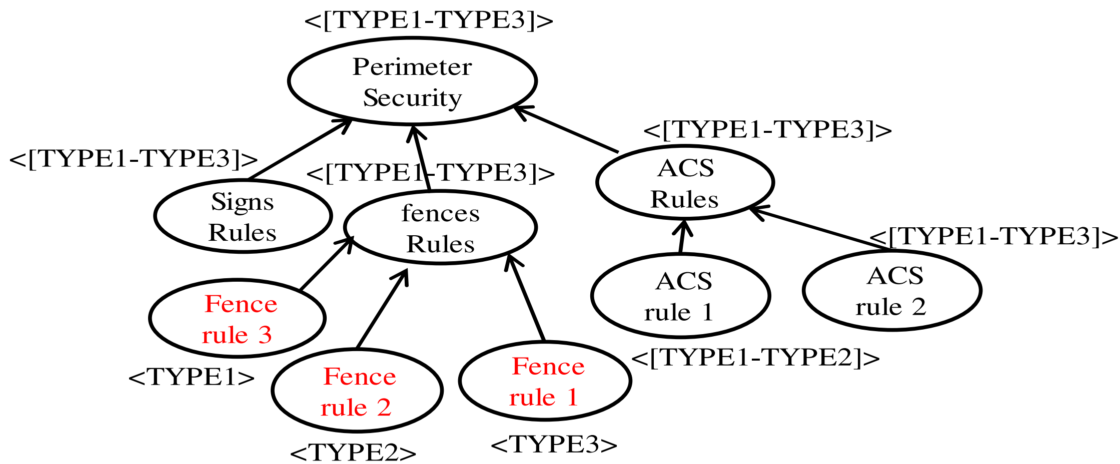
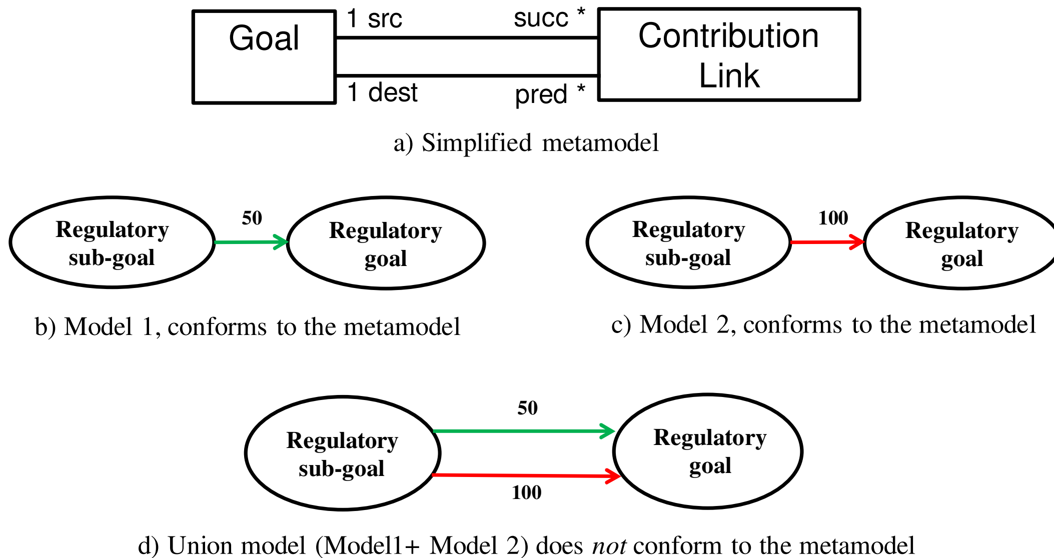


Figure 4 A union model for the model family in Figure 2

1.3. Other Challenges at the Metamodel Level

Capturing all model variants in a family (of one language) with a union model would have the potential of increasing analysis efficiency and minimizing maintenance problems. However, while creating a union model, modelers may face conformance violation issues with the metamodel of that given language. This is because, sometimes, the modeling language may not permit to capture all model variations with one model. For example, GRL limits the number of links between a pair of goals to 1, whereas the union model may need many. This is illustrated in Figure 5, where integrating two simple GRL models leads to a union model that is not a valid GRL model (i.e., not conforming to the GRL metamodel)

as the latter violates the constraint on the maximum number of links between a pair of goals.



a) Simplified metamodel

b) Model 1, conforms to the metamodel

c) Model 2, conforms to the metamodel

d) Union model (Model1+ Model 2) does *not* conform to the metamodel

Figure 5 Example (using the GRL language) of a union model that does not conform to the metamodel of the individual models in the model family

This issue is not limited to GRL but is also common in most modeling languages, where union models of a particular language may violate conformance with the original metamodel of that language. For instance, in UML class diagrams, two variations of a class may contain the same attribute with two different data types, and both data types need to be considered for that attribute (in the union model) because they may represent, e.g., different design alternatives.

The general problem of metamodel conformance violation that result from union models is illustrated in Figure 6. In such scenarios, we need to relax the language's metamodel (MM) into MM_U to ensure that the union model M_U is conformant, and hence, representable. A minimal relaxation is desirable in this context in order to minimize the required modifications to existing tools and analysis approaches.

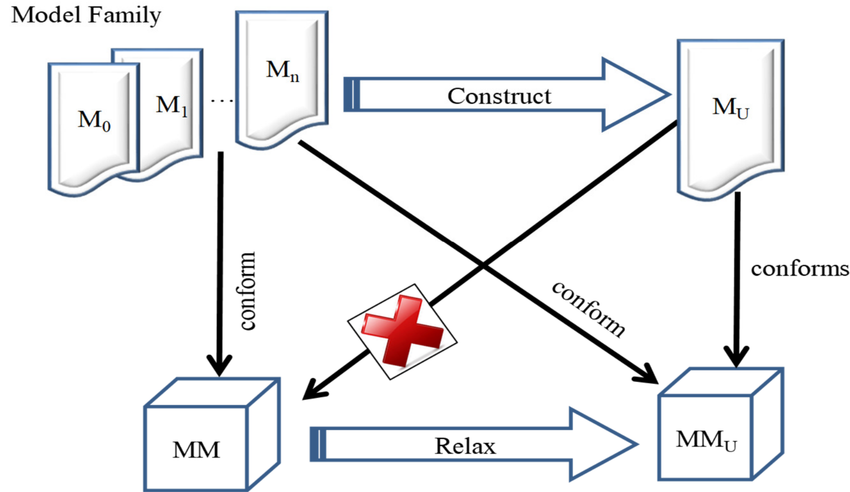


Figure 6 Model family-specific metamodel conformance problem

1.4. Thesis Objectives and Scope

This thesis aims to provide a modeling paradigm to model and reason about model families that vary/evolve over the space/time dimensions. In particular, it aims to address two correlated aspects at both model and metamodel levels. At the *model level*, the goal is to enable an efficient analysis of a group of models, *all at once*, using union models. At the *metamodel level*, the goal is to support the representation of union models (as valid instances of a metamodel) by inferring the metamodel of a model family from the *structure* of the metamodel of its members. In particular, the thesis aims to define a method that *minimally* relaxes the original metamodel constraints related to multiplicities of attributes and association ends.

At the *model level* (and for any given language), the scope of the thesis can be articulated as follows (see Figure 7):

- The formalization of models (resp. metamodels) as typed (resp. type) graphs.
- The encoding of the resulting typed graphs into propositional logic formulas combined with *annotations* on elements. The propositional encoding is performed manually in this thesis.
- The incremental construction of union models, using a language-independent union algorithm, from the set of annotated typed graphs. The resulting artifact is an

annotated graph, called Graph Union (G_U), which represents the union of all elements in all typed graph.

- Conducting efficient analyses at the level of G_U, where some of the analysis algorithms are implemented as general, language-independent algorithms at the G_U level, and other algorithms are language specific.

At the *metamodel level* (and for any given language), the scope of the thesis is characterized as follows (see Figure 7):

- Based on the structure of the type graph (TG) of a given language (i.e., the metamodel) and the evolution/variation behavior of models (captured in G_U), infer a relaxed metamodel TG' that supports G_U as a valid instance model.
- Transforming TG' back into MM' as well as transforming G_U back into M_U are out of the scope of this thesis.
- This work is not dealing with the more general problem of inferring a metamodel from a collection of *heterogenous* models that conform to different metamodels. Our work assumes that all members of a model family are *homogenous* (from the same language). In other words, the members conform to the same metamodel, as described in the model family-specific problem (Figure 6), which is the most typical context of model families. In addition, this work is still applicable to languages such as UML and URN, with many different types of diagrams but one metamodel.
- This work is not dealing with dynamic metamodel co-evolution upon model changes. This is because generating a new metamodel each time there is a new member added to the family is not practical, as this would imply developing new tools (for producing, analyzing, and transforming union models) each time.
- In order to relax metamodel constraints, we are looking for language-independent solutions, not for language-specific solutions (e.g., using metadata or user-defined links in GRL).

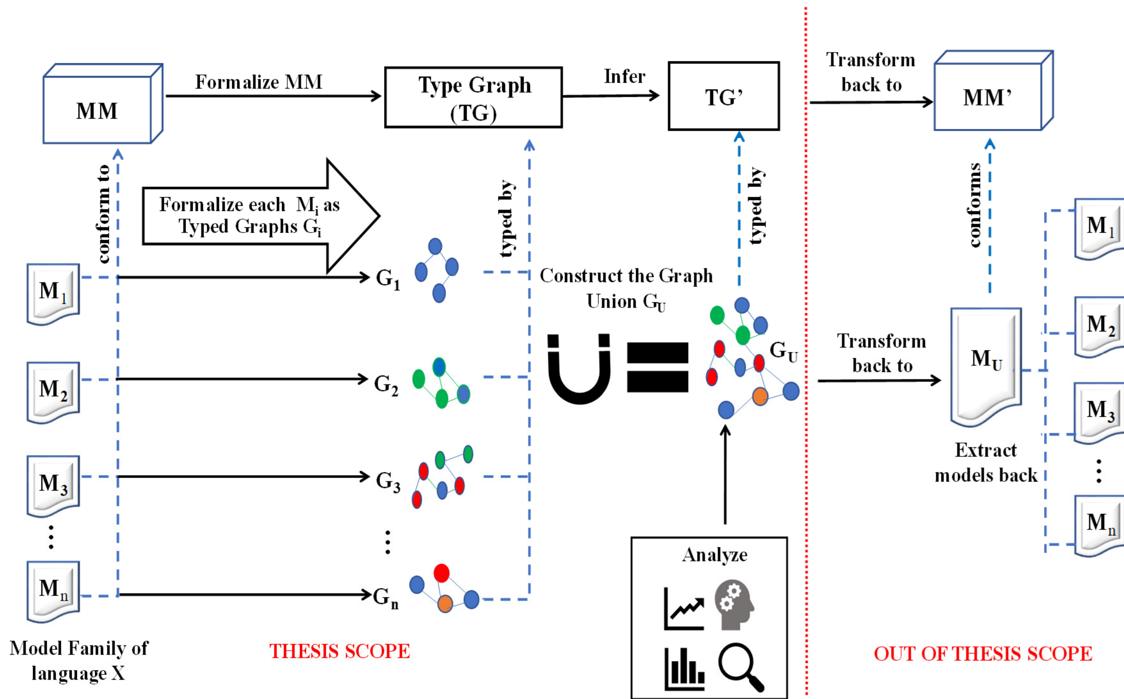


Figure 7 Scope of the thesis

1.5. Research Questions

This research aims to answer the following research questions:

- ***RQ1***: How can we represent the space/time-evolving model family with a union model M_U in a *compact*, *exact* and *self-explanatory* manner, while supporting *both* dimensions of variability?
- ***RQ2***: How efficient is reasoning and analysis with a group of models, all at once, using M_U, in comparison to the use of individual models?
- ***RQ3:*** Can there be a performance gain from adapting (or lifting) *existing* analysis techniques *specific* to a modeling language on a family of models, *all at once* using M_U, compared to analyzing individual models, *one model at a time*?
- ***RQ4***: How can we minimally relax a metamodel to support the representation of M_U as a valid instance model?
- ***RQ5***: How can we infer where relaxation is needed (i.e., *relaxation points*) in the original metamodel, for all potential model families of a language?

1.6. Methodology

Following the directions of March and Smith [13] and in order to contribute to the research community, this thesis aims to provide a *construct,* namely a new modeling approach that supports the modeling of space/time-evolving models in the context of model families. Such construct is proposed to address a specific *problem*, namely to enable a more efficient analysis of family members all at once, compared to the analysis of individual models, one model at a time, and also to support other types of analyses, using union models, that are otherwise infeasible or costly using individual models.

The underlying research methodology for this thesis is inspired by the design science research methodology, an approach towards research in information systems suggested by Von Alan et al.[14]. Following this methodology, the research started first with a literature review in the context of evolution and variability management of MBE artefacts (mainly models and metamodels). The purpose of the literature review is to highlight basic concepts, the current state of the art, and gaps in the literature in this context. Then, a conceptual characterization of the model family-specific context and the evolution of family members over time and space are identified (using inspiring analogies from other well-investigated domains), along with the challenges associated with this context. The problem is identified and scoped, and further illustrated through representative scenarios and examples. After building a knowledge base for the research, we characterize the requirements of the solution needed to tackle the identified problem. In particular, we propose a new modeling paradigm (using union models) to model and reason about families of models that change over space and time. We also identified the most crucial features of the proposed solution in order for it to dominate other existing approaches.

Finally, the feasibility of the proposed modeling approach is evaluated empirically, using several model families (or product lines) and several of their versions, across multiple modeling languages. The purpose of the evaluation is to assess the feasibility and usefulness of using union models to address the motivating problem.

1.7. Thesis Contributions

By addressing RQ1-RQ5, this research provides the following major scientific contributions:

- A language-independent, graph theory-based formalization of model families and union models (RQ1).
- A language independent algorithm (based on our formalization) to produce a union model from a set of models (in a compact and exact manner) of a given language (RQ1).
- A Spatio-Temporal Annotation Language (STAL) to support the representation of variability in model families (in space and time dimensions) and to facilitate reasoning about union models (RQ1).
- Improved efficiency of analysis and reasoning over a set of models, all at once (using the union model) compared to reasoning on single models, one model at a time (RQ2).
- Examples of analysis techniques adapted to support efficient reasoning about model families (RQ3).

The thesis also provides the following minor contributions:

- This thesis addresses the metamodel-level challenges associated with union models. In particular, it contributes a characterization of the requirements for minimally relaxing modeling languages (based on the structure of original metamodels) to support all potential union models of a language (RQ4).
- The thesis proposes two methods (*Evi-MeReFam* and *Anti-MeReFam*) to infer/anticipate locations where metamodel relaxations are needed (i.e., relaxation points) so that existing tools and analysis techniques be adapted *once* per language (RQ5).

1.8. Publications

This research has led, so far, to the following publications, directly extracted from this thesis:

- **Alwidian S.**, "Modeling Language Evolution for Model Family Support".
  In MODELS (Satellite Events). CEUR-WS, vol. 2019, pp. 492–495, 2017.

- **Alwidian, S.** & Amyot, D.: "Relaxing Metamodels for Model Family Support", in 11th Workshop on Models and Evolution (ME 2017), CEUR-WS, vol. 2019, pp. 64–70, 2017.
- **Alwidian, S.** & Amyot, D.: "Union Models: Support for Efficient Reasoning about Model Families over Space and Time". 11th System Analysis and Modelling Conference (SAM 2019), Munich, Germany, September. LNCS 11753, Springer, pp. 200–218, 2019.
- **Alwidian, S.** & Amyot, D.: "Inferring Metamodel Relaxations based on Structural Patterns to Support Model Families", 13th Workshop on Models and Evolution (ME 2019), Munich, Germany, September. IEEE CS, 2019.
- **Alwidian, S.** & Amyot, D. "On the Characterization of Model Families and Union Models". Poster, 11th Workshop on Modelling in Software Engineering (MiSE'2019), Montreal, Canada, May 2019.

Furthermore, at the early phases of my research, while I was exploring and learning about goal modeling, I had the following publications (the first two are in collaboration with the *Towards Cyberjustice* project at the Université de Montréal):

- **Alwidian, S.**, Amyot, D., & Babin, G. "Evaluating the Potential of Technology in Justice Systems Using Goal Modeling". E-Technologies: Embracing the Internet of Things (MCETECH 2017), LNBIP 289, Springer, pp. 185–202, 2017 **(Best Paper Award).**

- **Alwidian, S.A.**, & Amyot, D. "Towards Systems for Increased Access to Justice using Goal Modeling". 8th Int. Workshop on Requirements Engineering and Law (RELAW 2015), IEEE CS, pp. 33–36, 2015.

- Abdelzad, V., Amyot, D., **Alwidian, S.A.**, & Lethbridge, T.C. "A Textual Syntax with Tool Support for the Goal-oriented Requirement Language". 8th Int. i* Workshop (iStar@RE 2015), CEUR-WS 1402, pp. 61–66, Ottawa, Canada, August 2015.

1.9. Thesis Outline

The rest of this thesis is organized as follows:

- Chapter 2 presents a review of related work from the academic literature.
- Chapter 3 provides a characterization of the concept of model families and defines the proposed annotation language (STAL) suggested to support the representation of union models.
- Chapter 4 defines the graph-based formalization of model families and union models.
- Chapter 5 evaluates empirically the feasibility of using union models in supporting analysis and reasoning tasks more efficiently. The chapter defines three language-independent reasoning tasks and evaluates their performance using union models in comparison to using individual model.
- Chapter 6 presents the experimental validation conducted to evaluate the feasibility of using union models with more sophisticated analysis techniques that are *specific* to one modeling language (namely for goal modeling languages and their analysis techniques).
- Chapter 7 presents the second aspect of this thesis, namely the metamodel relaxation methods proposed to solve the problem of non-conformance introduced by union models.
- Chapter 8 concludes the main points of the thesis, with emphasis on the thesis' contributions, threats to validity, and future work items.

Please note the availability of many artefacts related to this thesis at http://bit.ly/2BW1zIG. These include model families and code for generating and analyzing models for Chapter 5 (in Python) and Chapter 6 (CPLEX).

Chapter 2. Literature Review

In this chapter, I discuss relevant background and related work from the academic literature. In Section 2.1, I introduce my review methodology. Sections 2.2 to 2.8 discuss variability management approaches in model based engineering, particularly in Model-Driven Engineering (MDE). Section 2.9 discusses approaches related to analysis of groups of models. Sections 2.10 to 2.15 report on metamodel-related evolution approaches. Finally, Section 2.16 provides a summary of the chapter.

2.1. Review Methodology

In this thesis, I investigate the work proposed in the literature for modeling variability in MDE, particularly in the context of model family, to support efficient analysis of model families over the space and time dimensions. In particular, I formulate my search queries according to the following questions:

- **Question 1**: What work was done in MDE that supports the capturing/modeling of variability in model families in space or time?
- **Question 2**: What work was done in MDE that supports the anlysis of models or model families?

In addition, since representing and capturing model families (using union models) is associated with challenges at the metamodel level (as discussed in Section 1.3), I further investiage:

- **Question 3**: What work was done in MDE that considers the evolution of metamodels to support the representation of model families?

The general query used to answer Question 1 is:

```
(represent OR capture OR manage)
AND (variability OR variation OR evolution)
```

```
AND ("model famil*" OR "family of models" OR SPL*
      OR "software product line*")
AND (MDE OR "model?driven engineering")
```

The general query used to answer Question 2 is:

```
(analyze OR reason)
AND ("model famil*" OR "family of models" OR "150% model"
      OR "150 percent model" OR "superimpos* model"
      OR "template model")
AND (MDE OR "model?driven engineering")
```

The general query used to answer Question 3 is:

```
(extend OR change OR modify OR expand)
AND ("meta?model" OR metamodel)
AND (MDE OR “model?driven engineering”)
```

Five popular and relevant search engines were considered for this review, including Scopus, SpringerLink, IEEE Xplorer, and the ACM Digital Library. Google Scholar was also included in order to cover a broader range of domains. Regarding Q3, I used a wide query to cover all metamodel evolution/change approaches in MDE, then, I used manual filtering to exclude irrelevant papers. For all questions, I also extended the initial selection with relevant articles cited in the papers selected so far (via snowballing).

Based on the above search method, I categorize the related work that were retrieved into the following research areas:

1) Research Area 1: Approaches related to *variability modeling* in MDE in space or time (Question 1). These approaches are discussed in Sections 2.2-2.8 and illustrated by the Venn diagram depicted in Figure 8.
2) Research Area 2: Approaches related to *analysis* of groups of models in space or time (Question 2), which are discussed in Section 2.9
3) Research Area 3: Approaches related to metamodel *evolution/relaxation* to support the representation of model families (Question 3), as discussed in Sections 2.10-2.15.

2.2. Research Area 1: Variability Modeling in MDE in Space or Time

The literature related to this research area is discussed in Sections 2.2- 2.8 as follows.

2.2.1 Overview

Throughout my literature review, I found that the following disciplines stands as core paradigms to which my thesis relates: *model-driven engineering (MDE)*, *software product line engineering (SPLE)*, and *version control (VC)*. These software engineering sub-disciplines are concerned with raising the level of abstraction of software development (through modeling), managing and organizing variability in a systematic way, and controlling the evolution of software artifacts, respectively. Furthermore, I observed that *variability in time* is tightly coupled with version control, where the latter is mainly concerned with managing program/system/product variations over time. On the other hand, *variability in space* is the main focus of SPLE, which is primarily concerned with configuration management in the space (or product) dimension. The following subsections briefly discuss these three paradigms. Furthermore, other interdisciplinary fields, which result from the intersections of these paradigms (as shown in Figure 8), are discussed subsequently. Finally, Sections 2.7 and 2.8 cover other variability management approaches based on models similar to union models and on goal models, respectively.

2.2.2 Model Driven Engineering

Model-Driven Engineering (MDE) is a software engineering methodology that uses *models* as primary artifacts to describe software systems at a higher level of abstraction [15]. A model can be defined using a general-purpose modeling language (such as UML) or with a domain-specific modeling language (DSML) [16] that provides modeling primitives to capture the semantics of a specific application domain (such as web-based languages). Most modeling languages are defined using metamodels, which in turn comply to a meta-metamodel (e.g., OMG's Meta Object Facility – MOF) [17], as illustrated in Figure 9, where the relationship between a model and its metamodel (at each layer) is defined as a conformity relation [18].

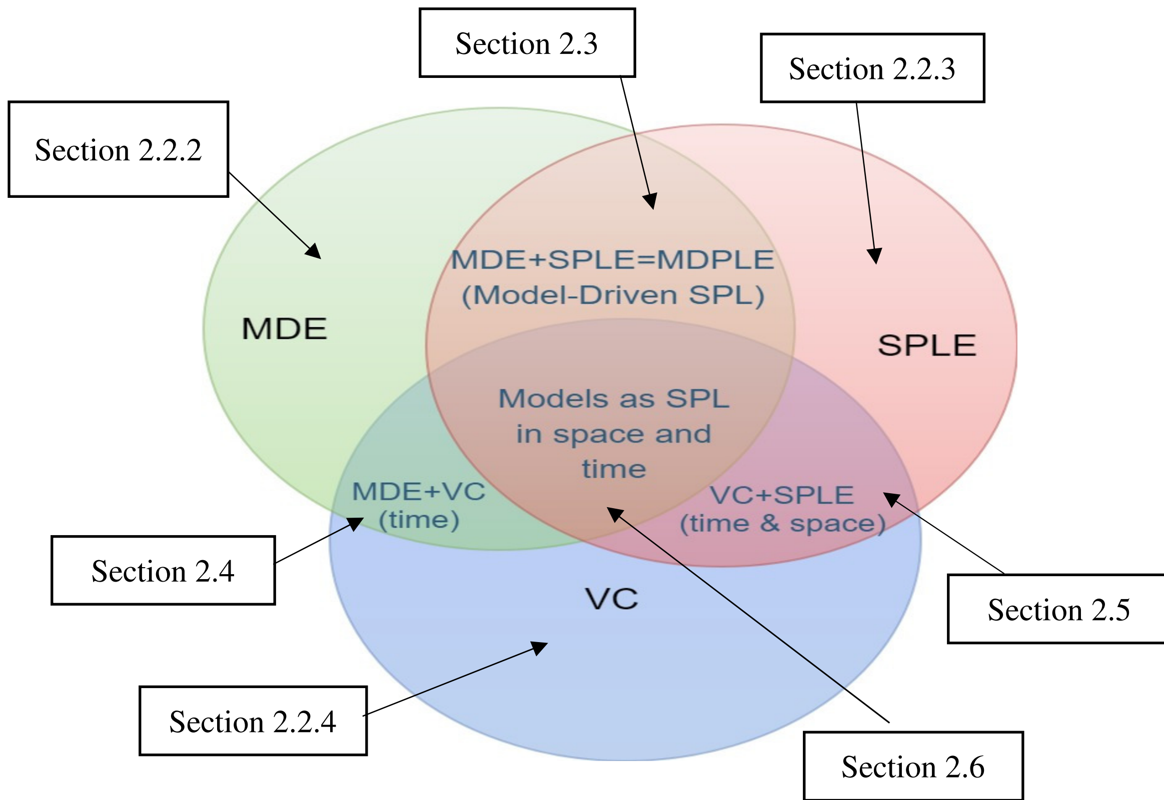


Figure 8 Organization of literature review for research area 1

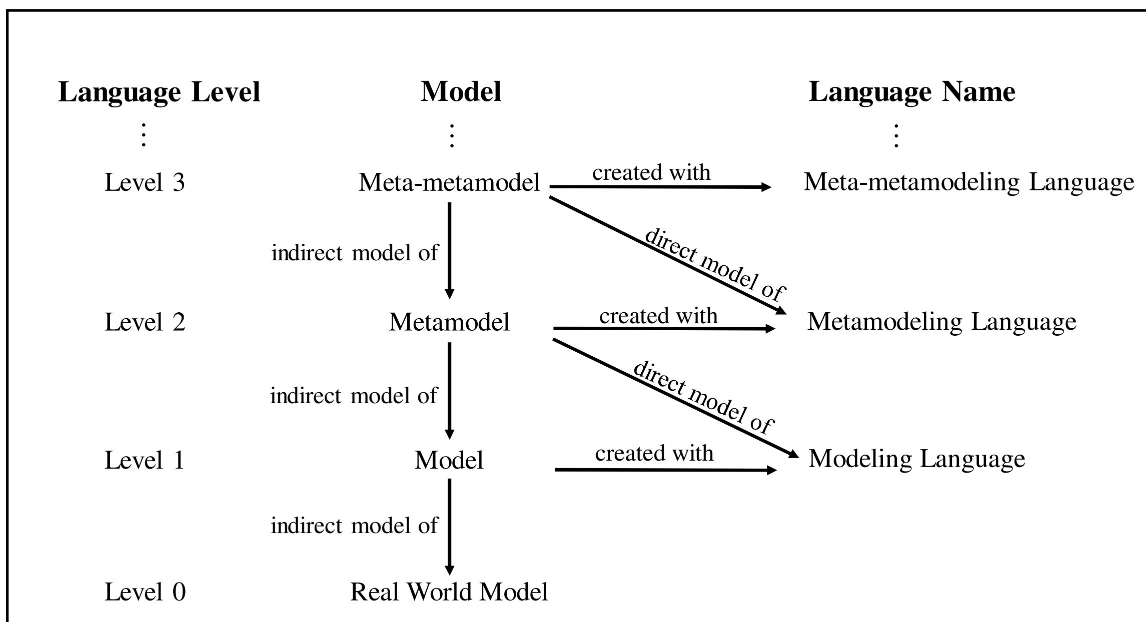


Figure 9 Modeling hierarchy in MDE [19]

The main goal of MDE is to enhance the development, maintenance and evolution of complex software systems by raising the level of abstraction from source code to models. In particular, the use of DSMLs enhances the quality and productivity of software development by allowing developers to focus on their essential tasks while the recurring engineering tasks are lifted to higher abstraction levels and automatically generated by transformations specified by domain experts [5][15].

2.2.3 Software Product Line Engineering (Variability in Space)

Software Product Lines (SPL) [20], known also as software product families or software families[3] [21][22] emerge as a paradigm shift towards the modeling and development of *families* of (software) systems rather than individual systems. There are several definitions in the literature for the concept of SPL. For instance, Northop [23] defined SPL as "*a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way*".

SPL engineering (SPLE) [3] endorses the principles of *mass customization* and *proactive reuse* of core assets. In particular, SPLE focuses on producing and maintaining multiple related software products (e.g., in automobiles or cellular phones), by exploiting *commonalities* (assumptions true for each family member) and managing *variability* (assumptions about how individual family members differ) between several software products [24][25]. Commonalities and differences among products are modeled and captured by *variability models*. One of the most commonly used variability models are *feature models* [4].

The SPLE process consists of two major phases: (1) *domain engineering* (or development for reuse), where core assets are developed and variability models are established, and (2) *application engineering* (or development with reuse), which focuses on the development or derivation of the final products (known also as *configurations*) using core assets and following customer requirements [3]. Both SPLE phases can be seen to be intertwined. This is because from one side, domain engineering feeds application engineering with core

[3] The term "software family" is used more frequently in Europe, whereas in North America, the term "software product line" is used more often as a synonym.

assets, and from the other side, the feedback provided by the application engineering is used to facilitate the construction/improvement of new/existing assets. Along the whole SPLE process, tool support for defining variability models, configurations, as well as for automatic product derivation are proposed in the literature.

Our work has strong conceptual resemblances with the domain of SPL engineering, which aims to manage software variants to efficiently handle families of software [3]. Although both of our work and the SPL domain have the concept of *families*, their *purposes* are different. In essence, the goal of SPL engineering is to plan for "proactive reusability", which means to strategically maintain a set of modelling artifacts (with high-level features) to exploit what variants have in common to derive or create new desirable products, some of which may not have been anticipated or foreseen. The goal of our work, however, is not to plan for reusability but to analyze families of *existing* models more efficiently using union models. In addition, SPL focuses usually on managing variability encountered in the space (or product) dimension. Our approach on the other hand focuses on modeling variability in both the time and the space dimensions.

2.2.4 Version Control (Variability in Time)

Version control (VC) is a technique for variability management in the time dimension, i.e., for evolution. VC becomes essential for software engineers to coordinate software changes inside a team and to control software evolution. Version control systems (VCS), such as *Subversion* [26], monitor and control models evolution by performing an iterative process consisting of check-out, modify, and commit phases. Through version control, collaboration between developers is enabled either by locks on currently modified resources (also known as pessimistic synchronization), or by merging possibly conflicting changes concurrently applied by several developers (known as optimistic synchronization).

Version control is mainly concerned with tracking and highlighting the changes that happen across models, and with calculating the final model based on the *differences* from the original model, without backward traceability to the source of the changes. This is different from our approach, where we calculate the union model by taking all elements that belong to all versions of models, with an additional feature that annotates elements to indicate to which version they belong.

2.3. Model-Driven Product Line Engineering (MDE+SPLE)

MDE has been increasingly used by organizations to effectively manage software product lines. The integration of both MDE and SPL leads to a paradigm knows as Model-Driven Product Line Engineering (MDPLE) [27][28].

Using MDE technology, SPLs can be planned, specified, processed, and maintained at a higher abstraction level where models, as first-class artifacts of MDE, can explicitly show both the common and varying parts of product lines.

2.3.1 Modeling Variability in SPLE

The term *variability*, in a common sense, refers to the tendency or the ability to change. There are several definitions of variability provided in the literature. For instance, according to Pohl et al. [3], it is the "*variability that is modeled to enable the development of customized applications by reusing predefined, adjustable artifacts".* For Bachmann and Clements [29], "*variability means the ability of a core asset to adapt to usages in different product contexts that are within the product line scope*". Last, but not least, Weiss and Lai [30] defined variability in SPL as *"an assumption about how members of a family may differ from each other".*

Central to the SPLE paradigm is the modeling and management of variability across product lines [31][32]. Variability is managed and modeled across two dimensions: (1) *variability in time*, which refers to the existence of different *versions* of an artifact that are valid at different times, and (2) *variability in space*, which refers to the existence of an artifact in different shapes or *variants* at the same time [3]. Variability modeling, in both dimensions, enables a company to select which version of which variant of any particular aspect is wanted in the system. Further discussion about variability characterization is provided in Chapter 3.

At the core of variability modeling in SPLE is the concept of *feature*, where features are expressed as variability points. According to Kang et al., the notion of feature is originally defined as *"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems"* [33].

In the domain of product lines, engineers and customers express the characteristics of a particular product in terms of what features this product has or delivers. To this end, it

is rational to use features to also express commonalities and variability among products in a product line. Czarnecki [34] adapted this idea to the SPL domain and defined a feature as *"a system property relevant to some stakeholder used to capture commonalities or discriminate among systems in a family"*.

A feature can have different roles in a SPLE process. For instance, in domain engineering, a feature is used to reflect evolution points of systems that correspond, e.g., to changing requirements. In application engineering, a feature is used to guide the creation and/or derivation of a product. This is achieved by carefully selecting a group of features, for which a mixture of parts of different components are involved [35].

To model variability in the SPL domain, and based on the notion of feature, a *variability model* should be used to capture commonalities and differences among product families. As an example of variability models, Kang et al. proposed to use *feature models* [33*]*. A feature model (FM) is a commonly used formalism to model variability in terms of optional, mandatory, and exclusive features organized in a rooted hierarchy, and associated with constraints and dependency rules over features[4]. FMs can be encoded as propositional formula defined over a set of Boolean variables, where each variable corresponds to a feature. FMs characterize the valid combinations of features as a *configuration*. A configuration defines, at a conceptual level, one product that can be extracted from the SPL.

FMs are deemed to be very useful formalisms to represent feature dependencies, to describe precisely allowed variabilities between products in a product line, and to guide feature selection as to allow the construction of specific products [4][36]. However, it is important to emphasize here that a FM is different from our proposed union models (M_U) in both formalism and usage. The differences between both artifacts can be summarized as follows:

1) A FM represents variability at an abstract "feature level", which is separate from the software artifacts (like a grammar of possible configurations), whereas M_U represents variability of all existing models at the "artifact level" itself.
2) While a FMs define all possible valid configurations of models along with constraints on their possible configurations, an M_U provides a complete view of the solution space that makes it explicit for modelers which particular element belongs

to which model, without necessarily modeling dependencies or constraints between elements of one model or across several models of a family.

3) The purpose behind using both artifacts is different: FMs are mainly used to ensure that the derived individual models are valid (through valid feature configurations), with the possibility of generating new models or products. On the other hand, M_{US} are proposed to perform analysis more efficiently, on a group of existing models, than on individual models.
4) Finally, an M_U enables the extraction of individual members of a model family by means of selecting particular time and/or space annotations, while FM enables model extraction by means of selecting valid combination of dependency rules and cross-tree constraints between features.

In fact, independently from the use of M_U, variabilities and commonalities (and their dependencies/constraints) among members of a model family can be represented as features in a feature model. A member of a model family can be produced or derived through the use of a valid feature configuration from the feature model.

2.3.2 Variability Modeling Approaches

Despite the ability of a feature model to capture a system's variation points in a concise manner, its elements are still propositional symbols whose semantics have to be provided by mapping them to other development artifacts [4]. Such mapping can be realized either explicitly, in a separate variability model, or implicitly, within the referenced artifacts. The following subsections discuss the three major approaches for variability modeling.

Annotative (or Negative) Variability Modeling

In MDPLE, annotative, or negative, variability approaches assume that a product line is represented in a form of a *multi-variant domain model (MVDM),* which is a base model (conforming to a specific metamodel) that contains every element that is used in at least one product configuration and, thus, subsumes every possible product [37]. MVDM is used to assist in product derivations from a particular SPL, where model elements are removed gradually to resolve a valid variant (hence the name negative variability).

In order to define which parts of the model have to be removed to derive a concrete product model, and to perform a fully automated product derivation, it is necessary to *annotate* or attach information to elements of the MVDM. In the literature, a common mechanisms to achieve this is to attach *presence conditions* to the MVDM elements [37][38], or to use stereotypes such as UML stereotypes [27][39]. Nevertheless, there are various approaches that differ in how presence conditions are expressed and physically attached; either internally (or implicitly) by extending the language itself, or externally through an explicit mapping between variability models (usually feature models) and domain models.

For instance, Gomaa [27] proposed a method referred to as *product line UML based software* engineering (PLUS). This method is mainly based on extending the UML modeling language with stereotypes attached to model elements. Examples of such stereotypes are *kernel* and *optional*, used to distinguish mandatory and nonmandatory elements, respectively. In addition, *default* and *variant* stereotypes are used to define variation points by providing their default and alternative variants. Variation points are modeled by a *UML-internal* specialization mechanism. This mechanism complicates both architectural decisions and product derivation, where products are derived by defining *views* that resolve configuration decisions. In this approach, the domain model and variability model are represented in the same artifact, and hence, they are not orthogonal.

There are other negative variability approaches in MDPLE, where the domain models and variability models are *intertwined*. Such approaches separate variability models from domain models, yet, the connection between problem space and solution space is realized within the domain model in an intertwined way. For example, Haugen et al. [39] proposed a separated-languages approach, where models are split and embedded in a generic variability-aware Common Variability Language (CVL) [40].

The Clafer [41] language mixes the concepts of UML class diagrams with feature models. In particular, a feature in Clafer is not only represented by a class, but also shares its structural features. This means that the feature model hierarchy corresponds to the object composition hierarchy defined in the intertwined domain model. Similarly, feature group constraints are equivalent to multiplicities of composition references.

In MDPLE, variability modeling approaches can also follow a *mapping-based* paradigm, where the MVDM and the feature model are kept *orthogonal*. Such orthogonality

is achieved by introducing a distinct *mapping model*, which interconnects the problem and the solution space. A mapping model, in its simplest format, assigns presence conditions (which are Boolean expressions on the variables defined in the feature model) to elements in the domain model. Examples of tools that are used to realize this mapping-based approach are FAMILE [42] and FeatureMapper [43].

The orthogonal variability modelling (OVM) approach [3] captures variability of product line artifacts using a separate variability model that is independent from the artifact model. Links from the variability model to the artifact model determine which model parts have to be removed for certain product variants. The variability modeling language (VML) was proposed by Loughran et al. [44] to specialize the ideas of OVM for architectural models.

Mapping-based variability modeling was extended by Czarnecki and Antkiewicz [36] to *template-based* negative variability. In this work, the authors proposed a general template-based approach for mapping feature models to concrete representations using structural or behavioural models. They use a model representing a *superimposition* of all variants (known also as the *150% model*), whose elements relate to corresponding features through annotations. In this work, the authors proposed to separate the representation of a product family into a feature model (that defines feature hierarchies, constraints, possible configurations) and a model template (or the 150% model, which contains the union of model elements from all valid template instances). Elements of a model template are annotated, and annotations are defined in terms of features from the feature model, and can be evaluated according to a particular feature configuration. Annotations used in this approach are *presence conditions* (PCs) and *meta expressions* (MEs). Typical PCs are Boolean expressions over a set of variables, each variable corresponding to a feature from the feature model, and they are attached to model elements to indicate whether they should be present in a template instance. If a PC is not explicitly assigned to an element of a model template, an *implicit presence condition* (IPC) is assumed to exist. The purpose of IPCs is to reduce the necessary annotation effort for the user. MEs are used to compute attributes of model elements.

Compositional (or Positive) Variability Modeling

In contrast to annotative variability modeling, compositional variability modeling, also known as positive variability modeling, starts from a minimal core that contains features that are common to all possible products. This core model is then extended by specific model fragments in order to compose specific applications, which correspond to new feature configurations.

Several approaches have been proposed in the literature to support the positive variability modeling paradigm, and among them is *model transformation*. In particular, positive variability can be carried out on top of some model transformations platforms, which consist of a core model, fragments, as well as model transformations. Fragments are defined as ordinary models, hence instantiated from a domain metamodel. Model transformations define how these fragments are attached to the core model. Product derivation in MDPLE consists in the selection of appropriate model transformations, which are then applied to the core model in a suitable order. An example tool that supports this approach is the Product Line Behavioral Synthesis (PLiBS) [38], which handles both structural and behavioral UML diagrams. In PLiBS, the integration of model fragments, identified by UML stereotypes, is controlled by model transformations.

Apel et al. [45] proposed an approach for product derivation based on positive variability. Different from the above approaches, Apel et al.'s approach hides the concept of transformations from the user. Fragments are merged into the core model such that sets of fragments, rather than transformations, have to be selected during product composition. To this end, a superimposition model is introduced as a model composition technique. In the superimposition model, elements are merged or unified based on their names and types. Then, the connection between fragments and the core model is made by copying or cloning parent elements where a fragment will be inserted.

Transformational (or Delta) Variability Modeling

Transformational variability modeling is a hybrid paradigm that combines both compositional and annotative methods. In this paradigm, model elements can be removed and added to resolve a variant. A well-known approach for this is the so-called delta modeling (or delta-oriented programming) [37]. Delta modeling consists of two parts: (1) the design of the core product, which consists of a set of feature selections that represent a valid product,

and (2) the specification of *delta modules* that describe changes to the core module. Each delta represents either an addition or a deletion of an element from the product model (see Figure 10 (c). A delta module is then associated to one or more features, and the associated deltas are applied to the product model whenever a feature is selected or deselected.

Examples of delta modeling approaches are the ones proposed by Clark et al. [46] and Schaefer et al. [47], where a set of systems are represented by a designated core system, and deltas are used to explicitly specify changes to the core system in order to obtain other system variants.

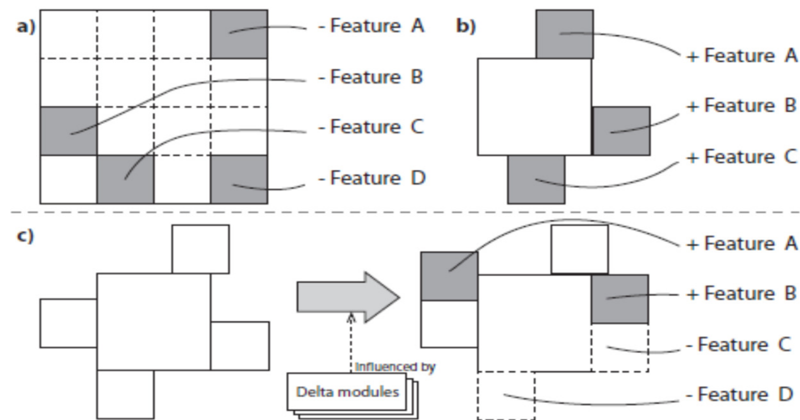


Figure 10 (a): Negative variability, (b): positive variability and (c) delta modeling (adapted from [48])

Summary

Among the three variability modeling approaches discussed above, annotative approaches are considered the closest to our approach. Both annotative approaches in SPL and our proposed work (represented using union models) consider the use of one generic model to represent all products of the product line, or all models in a model family, respectively. Hence, both approaches are deemed to be conceptually similar. However, they are still different in their purposes, their annotation mechanisms, and their reliance on feature models, as summarized below:

1) *Purpose*: The purpose of annotative approaches is to guide the derivation of concrete products from the 150% model, where elements are extracted only in case their presence condition evaluates to true given a particular configuration. Many unanticipated models can be extracted. On the other hand, the purpose of union

models is to facilitate an efficient analysis of group of existing models, all at once, as opposed to analysing individual models, one model at a time.

2) *Annotation mechanism*: In annotative approaches, *presence conditions* are usually used as annotations to indicate whether an element should be part of the derived product or not. Elements of M_U, on the other hand, are annotated with information about space and/or time, using STAL, to indicate the model variation and/or version that they belong to.

3) *Reliance on feature models*: In annotative approaches, presence conditions by definitions are Boolean expressions over a set of variables, where each variable corresponds to a feature from the feature model. This means that annotative approaches rely primarily on the existence of feature models [36]. This is a major difference from M_{Us}, where there is no need for a feature model to annotate a union model, since the latter is annotated with information about space and time that is inferred directly from model versions and/or variations in the model family (of course assuming that version numbers and variations are known a priori).

2.4. Model Version Control (MDE+VC)

In the last two decades, there has been an increasing interest in the literature about tracking and understanding differences between model versions in terms of the used modeling language, rather than through the textual representations of models, such as XMI [49]. This paradigm shift led to the emergence of the discipline named *model version control* (MCV) [50]. MCV involves two important sub-problems, namely *model comparison* and *model merging*. Surveying the literature about version control systems (such as EMF Store [51] or Odyssey VCS [52]) is however out of the thesis scope.

Model comparison, as the first phase in MVC, is concerned with comparing two versions of a model, and it involves two steps: *matching* and *differencing*. Two models are matched against each other to identify commonalities between models, which are recognized as correspondences between elements of the first version with matching elements in the second version [53]. Differences are then inferred from elements in one version that have no correspondence to elements in the other version. Several approaches exist in the literature to obtain matches and differences, including *operation-based* approaches. In such

approaches, the edit operations performed by users are recorded, so that a precise edit script is available for pairs of versions each time an edit is carried out. Operation-based versioning has been implemented in CoObRA [54] and EMF [55]. To facilitate comparison and matching of two models, many modeling tools (such as EMF Compare [55]) assign universal unique identifiers (UUIDs) to model elements. The use of UUIDs serves as a precise and simple criterion for identifying correspondences between model elements. If neither edit scripts nor UUIDs are available, heuristic matching takes place, where the edit operations carried out on model versions are deduced in a state-based manner. An example of these heuristics are similarity-based matching algorithms [55][56], which aim to define a set of correspondences between elements that have the highest possible degree of similarity. The result calculated by these algorithms does not necessarily reproduce the actual editing history.

After calculating the matches between two model versions, calculating *differences* becomes simple. That is, if v1 and v2 are two versions of the same model, elements belonging to v2 that have no matching elements in v1 are considered as insertions. Deletions, on the other hand, can be inferred from elements that are part of v1 but not of v2.

In MVC, it is vital to identify and record differences between model versions for two reasons: (1) to help users understand changes performed by themselves or by others and (2) to check and verify internal consistency of models. To this end, supporting visualization for model differences is required. Several approaches described in the literature, e.g., the approach proposed by Kehrer et al. [57], focus on lifting up the semantical level of differences to describe changes as a sequence of refactoring rather than in terms of low-level edit operations.

Model merging: According to Conradi and Westfechtel [58], model merging approaches can be classified into three categories.

1) *Raw merging*, where a composite change (i.e., a sequence of primary change operations) is applied to a particular input version that does not incorporate these changes yet.
2) *Two-way merging*, where two alternative versions are compared side by side to identify differences. If any difference is detected, a user intervention is required to decide on what has to go into the merged version, *m*.

3) *Three-way merging*, where the merge decision is partially automated by checking a common reference version, called the *base version (b)*. The general concept of three-way model merging is as follows: from a base version *b* of a model, two alternatives a_1 and a_2 are derived by applying several change sequences c_1 and c_2, respectively. Then a merged model *m* is constructed by integrating all element additions and removals.

The three types of merging are illustrated in Figure 11.

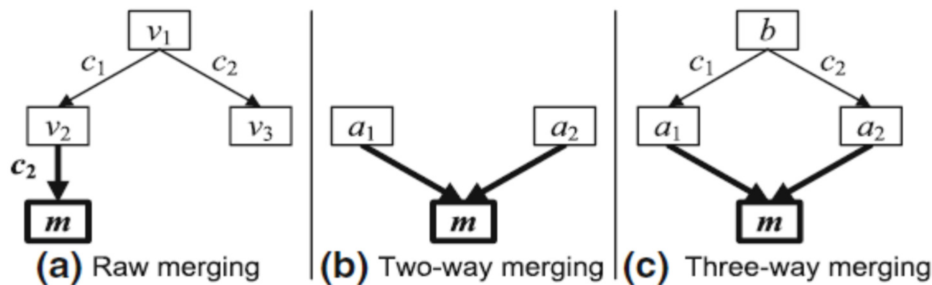


Figure 11 Types of merging (adapted from [59])

In the literature, there are different approaches to realize the three-way merging, namely *state-based* merging and *operation-based* (or *change-based*) merging, as illustrated in Figure 12.

In *state-based merging*, the states of versions a_1 and a_2 and the base version *b* are taken as inputs. The versions are compared pairwise along three ways. A merged version *m* is created that includes elements from a_1 and a_2 with the intent to merge the changes that have been applied concurrently to *b*. If there are differences between a_1 and a_2, the base version is inspected to figure out which changes have been performed on each branch. If changes have been performed on both branches, a conflict is reported. If, on the other hand, a change has been performed only on one branch, then it is applied automatically to *b*. Westfechtel [59] proposed a pure state-based three-way model merging algorithm. The algorithm applies context-sensitive and context-free merge rules to construct a merged model graph. To determine the objects to be included in the merged model graph, the authors used set-theoretic formulas. This algorithm has been implemented in the BTMerge tool [60], which performs interactive and consistency-preserving three-way merge for EMF-based models.

Alternatively, *operation-based* or *change-based merging* [61] takes the base version b and the *changes* $c1$ and $c2$ from the base to the alternative versions as inputs. The change operations are combined in an order-preserving way to produce a *merged change* cm which is used to create m from b (Figure 12. b).

Alanen and Porres [62] proposed a three-way change-based approach to calculate the difference between two models (represented as a sequence of operations), and then extend the difference calculation to form a union algorithm. The union algorithm calculates the union of two models based on their differences from a given original model (or a base model), where two separate modifications are made to a base model b, and the union algorithm combines both differences into one model by interleaving the operations in the latter difference with the former difference. For example, given a base model M_{base} and two alternative model versions M_1 and M_2, the union of these models, denoted as M_{final}, is calculated as: $M_{final} = M_{base} + (M_1 - M_{base}) + (M_2 - M_{base})$.

Altmanninger et al. [63] proposed a change-based three-way merge tool, where EMF Compare is used to obtain matches between models, and then changes $c1$ and $c2$ are inferred in the form of graph modifications. Conflicts are also made visible in the form of model annotations, which allow users to resolve them non-interactively.

The major difference between our proposed approach and the *three-way merge* paradigm is that the latter is concerned with tracking changes (or deltas) that happen across models (using a *diff* algorithm), and calculates the final model based on the *differences* from the original model. Our union algorithm, however, calculate the union model by taking all elements that belong to all versions of models, where elements are distinguished by means of annotations to indicate to which model version they belong.

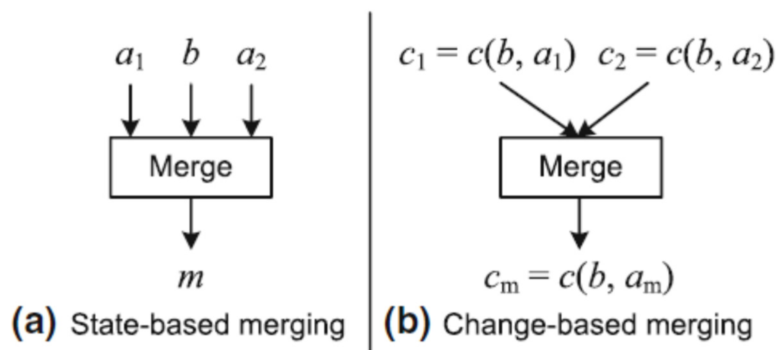


Figure 12 State-based and Operation-based model merging (from [59])

Model merging is also applied in distributed model-based development, where models are constructed and manipulated by distributed teams, each working on a partial view of the overall system. In this context, management of models relies significantly on putting together models coming from different sources, a process referred to in the literature as *merge* [64]. However, model merging in distributed environments is challenging due to several reasons: first, models usually overlap, such that they could refer to the same concepts in the requirements or design of a system, but the overlapping concepts may be presented differently in each model, and hence, models may contradict one another. In addition, modelers may use different modeling notations, or use the same notations inconsistently. Furthermore, and most importantly, models may evolve into different versions, so that merges may need to be recomputed if the source models are updated. Examples of model merging approaches include static and dynamic UML models [65], requirements models [66][67], program variants [68], and multiple varieties of reactive systems [69][70]. Brunet et al. [64] proposed a framework for comparing different model merging approaches by proposing a number of merge-related operators and discussed their desired properties. In particular, model merging was considered an algebraic operator over models and model relationships.

In these approaches, underlying assumptions and modeling formalisms differ from our proposed work. For instance, most of these approaches require that only consistent models can be merged, which is not the case in this thesis' union models. Hence, inconsistent models must be repaired prior to merging. In addition, the purpose of merging models in these approaches is to show differences between models rather than to put them together and analyze them, all at once.

2.5. Software Product Line Version Control (SPLE + VC)

Sometimes variability in time (which is the focus of VC) and variability in space (the subject of SPLE) may overlap and need to be considered together as two interrelated dimensions of variability, rather than being addressed in isolation from one another.

To elaborate further, integrating SPLE with VC is necessary to support collaborative SPLE and to facilitate a systematic change management of all artifacts that are part of

an SPL, such as the platform (i.e., the core assets of different products), the variability models (e.g., feature models), as well as the mapping between them.

In the literature, however, there are few version control systems that support software product lines. For instance, among other researchers, Schmid et al. [71] proposed a dedicated approach for software product line version control (SPLVC), which takes into consideration the fact that product lines are more long-living, combined with the numerous artifacts (i.e., products) that change continuously over time, or that change in terms of features. This approach goes beyond directly applying the state-of-the-art VCS to a software product line.

In the area of variability management in space and time, the work related closest to ours is the work of Seidl et al. [72][73]. The authors proposed an integrated approach to manage variability in space and time in software families (i.e., SPL) and software ecosystems (SECOs). In this work, Hyper Feature Models (HFMs) [72], as variability models, are used to represent features with versions and are combined with delta modeling as a variability realization mechanism. This integration allows the derivation of concrete software systems (i.e., products) from an SPL or SECO, configuring both functionality (in the product dimension) as well as versions (in the time dimension). In order to allow the definition of valid combinations of versions of features, a version-aware constraint language is also provided.

Mitschke and Eichberg [74] proposed a feature-driven versioning approach, where they extend each feature in a feature model with two version numbers to signal the revision of the sub-branch beneath the feature (feature logical version) and that of the associated realization assets (feature container version). However, there is exactly one of these versions for each feature so that feature versions cannot be used as configurable units.

Cardinality-based feature models were proposed by Czarnecki et al. [75], who extended the original feature modeling notations with feature attributes. Such attributes represent versions of features with all possible intervals in the domain of that attributes.

In [76], Dhungana et al. proposed an approach to support product line evolution by organizing feature models of large-scale product lines as a set of interrelated model fragments defining variability of particular parts of the system. In particular, the authors proposed to divide up the feature model into several sub-trees, and references to elements

contained in fragments are established by means of a placeholder mechanism. The approach allows for semi-automatic merging of fragments into complete variability models. As different fragments are versioned in isolation, many SPL evolution problems are avoided.

Although the above-mentioned approaches address the issue of capturing variability in space (through SPLE) and time (through VC), they are still *different* from our work in the following aspects:

1) In SPLVC, variability in space is handled from an SPL point of view; this means that the use of variability models (usually feature models) is mandatory, and the purpose of handling variability is mainly for extracting or generating valid products (according to the valid feature configuration as already discussed in Section 2.2.3). Unlike in the SPL paradigm, our approach does not necessarily require any variability model, since variability is intrinsically captured in M_U itself.
2) In SPLVC, variability in time is addressed from the perspective of VC; this inherently leads to the differences discussed already in Section 2.2.4

2.6. Models as SPL in Space or Time

Section 2.3 discussed model-based SPLs, where the members of an SPL are specified and represented in the form of models. This section surveys approaches with the opposite perspective, which in particular view and specify a given family of models as an SPL.

Font et al. [77] proposed an approach called *Model Family to SPL*. This approach automates the formalization of the implicit variability that exists among a given set of similar models into an explicit variability as an SPL. The variability is made explicit in that approach using CVL [5]. In particular, the Model Family to SPL approach takes a family of models (modeled in any DSL conforming to MOF) as input and generates a CVL-based SPL, where commonalities and variabilities among the model family are explicitly defined. The model commonalities are formalized as a base model and variabilities are specified as placements over the base model and replacements in a model library. The generated SPL of models can be further evolved to include new products (i.e., models) and it is also capable of generating all the products back from the given model family.

Avila-García et al. [78] proposed a domain specific transformation language called *Model Template Transformation Language* (MTTL) in order to facilitate the creation of SPLs to manage and specify model families. The authors provide an example of representing a model family of state machines as a model product line. The use of MTTL helps SPL engineers to specify rules of model template specialization in a product line of models.

In their subsequent work, and inspired from Czarnecki and Antkiewicz's work [36], Rebull et al. [79] used *model templates* as a solution to maintain use case model families as software product lines of models. The created model template contains all variants of use case models in a superimposed form. To extract concrete family member (i.e., individual use case model), a model transformation automatically specializes the model template by deleting object relations from the model template. The specialization is carried out following a set of selected features (i.e., optional and alternative) that are specified in a feature model which characterizes the model family.

In [80], Martinez et al. proposed the *Model Variant Comparison* (MoVaC) approach. The main goal of this approach is to compare a set of model variants to identify commonalities and variability as features, where each feature consists of a set of atomic model elements. MoVaC also focuses on the visualization of the identified features using a graphical representation where common and variable features are explicitly presented to users.

Palmieri et al. [8] proposed a tool-supported approach that integrates the Goal-oriented Requirement Language (GRL) and feature models to handle and represent regulatory goal model families as software product lines. This is achieved by annotating a goal model with propositional formula related to features in a feature model. The authors provided techniques to check consistency of the resulted SPL as a whole and ensure the derivation of valid goal models associated to a feature model configuration. However, Palmieri et al.'s work did not consider the evolution of goal models over time, and did not introduce union models.

The above-mentioned approaches are close to ours in the sense that they deal with the concepts of *model families* and *model templates* (or model superimpositions), where the latter contain all model variants of the family in a superimposed form, and this is

typically similar to the concept of union models. Nevertheless, clear differences can be spotted between these approaches and ours, as follows:

1) All of the above approaches specify and maintain a given model family as a software product line of models. This leads to the main point discussed in Section 2.2.3 about the differences between SPL paradigm and our approach.
2) Unlike union models, the construction of a model template depends solely on feature models.
3) To obtain or extract a concrete family member (i.e., an individual model) in the above-mentioned approaches, a model transformation specializes the model template. The specialization is carried out following a set of selected features that are specified in a feature model which characterizes the model family.
4) In these approaches, annotations are defined in terms of features from the feature model, and can be evaluated according to a particular feature configuration. Possible annotations are presence conditions (PCs) and meta expressions (MEs). Union models use a different annotation mechanism, where elements are annotated with space and time information using STAL.

2.7. Variability Management Using Union Models-like Artifacts

This section surveys the approaches that use artifacts which are conceptually similar to union models. The most popular artifact is the *150% model*. The concept of 150% model was first introduced and used in the context of SPL engineering to manage variability and to facilitate the automatic derivation of a specific product variant (called 100% model) and/or for model-based testing of product lines [81].

Several approaches exist in the literature that made use of 150 % models. These approaches agree on a general definition of a 150% model. We infer this agreed-upon definition as: a *super model that contains the union of all sub models, from which valid member models can be extracted*. However, each approach used a slightly different term to refer to the concept of 150% model, resulting in a set of synonyms for that concept such as: a

150 percent model, a complete model, a superimposition model, a product generic model, generic model, a reusable model, a template model, or a family model[4].

For instance, Grönniger et al. [82] used a 150% model as a complete function net that describes the whole logical system architecture of automotive systems. Such model consists of all features in all possible variants. The authors referred to the model as a "complete model" or "150 percent model". From a 150 percent model, variants can be chosen by parameterization (i.e., using feature models). They refer to the realization of a single logical architecture as the 100% model.

In [81], Polzer et al. described a *negative variability* approach to automate the product derivation process through mappings between features and implementations so as to automatically derive implementations from a particular feature configuration. In this approach, a 150% model was constructed such that it contains all implementations of all potential variants of a product line. This 150% model was used to facilitate the process of product derivation from model-based product line.

Weißleder and Lackner [83] proposed two approaches for model-based test generation applied to product lines: a top-down and a bottom-up approaches. The authors intended to generate tests for an online shop product line. In order to do so, the model for test generation is linked to the feature model. Then, the authors use a 150% state machine model to describe all variation points in the state machine (i.e., the possible behavior of the system under test), and link features of the feature model to these variation points. By linking feature models with a 150% state machine model, the authors provide the infrastructure for the two approaches of model-based test design for product lines.

In [84], Cichos et al. proposed an SPL test suite generation algorithm that uses model-based, coverage-driven testing techniques to derive a small test suite from one 150% test model of the SPL such that a given coverage criterion is satisfied for the test model of every product. 150% test model (also referred to as super test model) contains all test models of an SPL as special cases [82]. By using the 150% test model, it is possible to determine if a created test case is executable on more than one product.

[4] A *family model* is a synonym of the 150% model, which is different from the *model family* concept introduced in this thesis. In particular, a family model is used to represent a model family in one generic model.

A tool chain called *Model-based Software Product Line Testing* (MoSo-PoLiTe), was proposed by Oster et al. [85]. This tool is a combination of model-based testing and combinatorial testing for SPLs. The testing tool involves many tasks, the first one making use of the 150% model to form the reusable test model. This model includes all commonalities and variabilities of the entire SPL, therefore, all information required to test each possible product of the SPL is available.

To summarize, since a 150% model of an SPL architecture constitutes the basis for modeling system configurations with all possible (and different) views/variants, it is of particular importance in variability-rich, product line domains (such as the automotive domain), especially when it is hard or unfeasible to generate and test every possible variant configuration, individually, of the product line. Therefore, the verification and testing of a (software) product line can be enhanced by using a 150% model that integrates all variants. However, a disadvantage of using a 150% model is the complex logical architecture, particularly when it comes to member extraction. The more variant the configurations, the more complex the 150% model, and the harder it becomes to extract a particular member model. Up to this point, this issue is still an open research challenge. One potential solution (proposed in this thesis) is to annotate/tag the various elements of the union model (which is equivalent to the 150% model) with information about version numbers. This way, elements of a particular model version can be extracted using the number of that version.

2.8. Variability Modeling using Goal Models

Recently in the literature, there has been much effort devoted to variability modeling using goal models. Lapouchnian and Mylopoulos [86][87] proposed a framework for modeling and analyzing domain variability for goal models. To distinguish elements that need to be visible in the model, the authors use contextual tags to label model elements. A Boolean variable is assigned to each tag, identifying whether a tag should be active or not. In addition, the authors in [86] propose an algorithm to extract parts of the goal model that are dependent on the context variability. Furthermore, they extend the *i** notation to represent and support variations in goal models [87].

Ali et al. [88] extended TROPOS with variation points to present contextual goal models, where the context may influence the choice among the available alternatives. The

authors also used tagging with conditions on goals and links (decomposition, dependency, contribution). However, instead of capturing families of related goal models, the work of Ali et al. [88] focuses more on describing runtime adaptation based on a logic-based representation of goals and conditions.

Goal-oriented languages have also been used to support feature models for software product lines (SPL). In particular, Silva et al. [89] proposed an extension of *i** that enables modeling common and variable feature of SPL with cardinalities using tasks and resources of a goal model to capture features. Borba et al. [90] have conducted a comparison between existing goal-oriented techniques for feature modeling in SPL.

Mussbacher et al. [91] proposed an SPL framework based on Aspect-oriented URN (AoURN) that allows capturing features and reasoning about stakeholders' needs. They mapped SPL concepts into AoURN concepts and applied their framework to the Via Verde SPL case study to evaluate the proposed solution. Yu et al. [92] also proposed a tool-based method to create feature models from a goal model.

Aprajita et al. [93] and Mussbacher [94] extended the metamodel of GRL to document explicit changes (additions/deletions) of model elements to specific versions of a metamodel. Although a model family can then be captured, this approach is specific to one language and currently incomplete in the kinds of changes to versions it can accommodate.

Grubb et al. [95] introduced the concepts of "dynamic intentions" into goal models to model alternatives on multiple time scales. The authors proposed a tool-supported method for specifying changes in intentions over time which uses simulation for asking a variety of 'what if' questions about models that evolve over time.

To support variability/variation of models in *a model family,* there has been few approaches created to support families using modeling languages such as GRL. For instance, Shamsaei et al. [9] proposed an approach that allows modelers to define a generic goal model family for all types of organizations in a legal compliance domain. They tagged model elements with information about organization types to specify which ones are applicable to which family member. The solution is formalized as a profile for GRL, with stereotypes, well-formedness constraints, and a modified analysis algorithm.

To the best of our knowledge, none of the approaches discussed above (except in [93][94] and [95]) support the evolution of models over time, nor do they document/model

that evolution in an artifact like a union model. In addition, those approaches do not take into consideration the case of metamodel-conformance violations that might be caused by model families (as model families may could violate the metamodel's constraints, and therefore, be invalid instances of metamodels).

2.9. Research Area 2: Product Line Analysis

This section focuses on surveying analysis approaches related to product lines, being families of (software) products, rather than surveying the classical analysis approaches of individual models.

In the last decade, the literature has witnessed a number of analysis approaches tailored to software product lines. The principal idea of such approaches is to exploit knowledge about commonalities and variability among products in a product line as well as their features to systematically reduce analysis effort. Existing product-line analyses are typically based on standard analysis methods, such as model checking, type checking, testing, static analysis, and theorem proving.

In this thesis, I follow the methodology introduced by Thüm et al. [96] to classify existing product-line analyses based on how they attempt to reduce analysis effort (referred to as the *analysis strategy*). In particular, I distinguish three analysis strategies, namely product-based, family-based, and feature-based analyses, to indicate whether the analysis is applied to individual products, the whole product line, or features, respectively. The following subsections discuss each one of these strategies.

2.9.1 Product-based Analysis

According to Thüm et al. [96], *product-based analysis* is defined as follows: "An analysis of a software product line is *product based* if it operates only on generated products or models thereof, whereas the variability model may be used to generate all products or to implement optimizations. A product-based analysis is called *optimized* if it operates on a subset of all products (a.k.a. *sample-based analysis*) or if intermediate analysis results of some products are reused for other products; it is called *unoptimized* otherwise (a.k.a. *exhaustive*, *comprehensive*, *brute-force*, and *feature-oblivious analysis*)".

In light of this definition, to perform a product-based analysis, products of a product line need to be generated or extracted individually, and then analyzed using existing analysis techniques. One naïve option to achieve this is to extract and analyze *all* products in a brute-force manner. An alternative option is the unoptimized product-based analysis [97][98]. The idea of such approaches is to consider a smaller number of products as a sample (usually based on some coverage criteria such as pair-wise [97] or t-wise [98]), such that correctness or other properties of the entire product line are considered satisfiable by this sample.

Several approaches were suggested in the literature to perform product-based analysis over SPL for different purposes and applications. One example is *type checking,* as proposed by Apel et al. [99], Istoan [100] and Buchmann and Schwagerl [101]. Moreover, product-based analysis was applied for *model checking* by Katz [102], Cordy et al. [103], and Fantechi and Gnesi [104] as well for *theorem proving* by Bruns et al. [105].

Product-based strategies, in general, are simple and practical, and they can be applied without adapting tools or using new techniques. However, a shortcoming of these strategies is that they require to generate all products of a software product line, which is often infeasible as the number of generatable products is up-to exponential in the number of features. In addition, analyzing each individual product separately is inefficient and involves redundant computations, due to commonalities between products. This is where our approach excels over these approaches, as a union model is used to analyze models in a family, all at once.

Regarding the unoptimized product-based analysis, since only a small number of products is used as a sample, and only those products are analyzed, then this strategy reduces the number of derived products. However, the sampling of products in inherently incomplete and leads often to an inefficient development and analysis of SPLs. This is also where union models achieve better performance in terms of improving the analysis efficiency.

2.9.2 Family-based Product Line Analysis

To reduce the redundant computations associated with product-based analysis, an alternative strategy for product lines analysis is to consider domain artifacts such as feature

modules instead of generated artifacts (i.e., products), and hence achieve more efficient analysis.

In the literature, analysis strategies that operate on domain artifacts and their valid combinations, as specified by a variability model, are referred to as *family-based analysis*. According to Thüm et al. [96], "an analysis of a software product line is family based if it (a) operates only on domain artifacts and (b) incorporates the knowledge about valid feature combinations".

In family-based analysis, the variability model is represented as a logic formula to enable analysis tools to reason about all valid combinations of features (e.g., SAT solvers can be used to check whether a particular property or method exist in all valid feature combinations). In other words, the goal of family-based analysis is to analyze domain artifacts and variability models, from which we can conclude that some intended properties hold for all products. To perform family-based analysis in practice, all features are usually merged into a single virtual product (also known as *metaproduct*) which is not necessarily a valid product per se (due to optional and mutually exclusive features) [96].

Family-based strategies have been proposed by several authors for *type checking* of software product lines. Examples of such approaches includes the work of Thaker et al. [106], Apel et al. [107] and Kolesnikov et al. [108]. More recently, family-based *static analyses* for software product lines, and data-flow analyses in particular, have been proposed in the literature. Ribeiro et al. [109] was one of the pioneers who proposed the first family-based static analysis. Their goal was to support product-line development and prevent errors up-front, rather than for product-line verification. Tartler et al. [110] proposed a family-based static analysis for defect detection in the Linux kernel. They analyze if code blocks surrounded by *#ifdef* directives are contained in any product (i.e., dead) or contained in all products that contain the parent block (i.e., undead). Furthermore, static analyses have been proposed so as to focus on analysis that is specific to product lines only (known as *family-specific analyses*), and do not scale for analysis of single systems. Examples include the work proposed by Adelsberger et al. [111] and Sabouri and Khosravi [112]. In contrast to family-specific analysis, other family-based static analysis approaches have been proposed to scale existing static analyses from single systems to product lines. Midtgaard et al. [113] showed how to systematically lift static analyses from single-system

engineering to product lines. Brabrand et al. [114] demonstrated how to transform any standard data-flow analysis into a family-based data-flow analysis.

Another category of family-based analysis is the *family-based model checking*. The idea of this type of analysis is to analyze each product/model of the product-line with respect to the variability model and one or more given properties. In particular, a model checker analyzes whether a given property is fulfilled by all products. If not, the model checker usually returns a propositional formula specifying those products that violate the property [115]. The majority of approaches for family-based model checking apply abstract model checking (i.e., they do not operate directly on the source code of systems, but on an abstraction of a system), such as the approaches proposed by Beek et al. [116], Classen et al. [117] and Cordy et al. [118].

Family-based analysis strategies and the use of union modes to analyze model families have common advantages and disadvantages. The advantage of both approaches is that they do not require every individual product to be generated and analyzed. This in turn avoids redundant computations across multiple products, and hence, increase the analysis efficiency. In addition, both approaches save effort to analyze groups of models, since the analysis here is not proportional to the number of valid feature combinations. On the other hand, the disadvantage of both approaches is the that analysis techniques for single products/models cannot be used as is, and usually, they need to be lifted or extended to work well with groups of models.

Despite that family-based strategies, like union models, analyze groups of products, both approaches are still different in the sense that most of the family-based strategies intrinsically depend on variability models (a process known as the *"early variability-model consideration"* [96]) to guide the analysis process. Hence, changing a small set of features usually requires analyzing the whole product line all over again [119]. This issue becomes particularly serious for large product lines with many features, and whenever product lines evolve over time. This issue is not faced in our approach, first, because there is no reliance on feature models, and second because union models are meant to reduce the overall analysis effort (by avoiding redundant computations) for large groups of models that evolve over space and time.

2.9.3 Feature-based Analysis

A third strategy for product lines analysis is the *feature-based analysis*. Here, the assumption is that certain properties of a feature can be analyzed modularly, without reasoning about other features and their relationships. Hence, all domain artifacts that fulfill a particular feature are analyzed in isolation, without considering other features or the variability model [96].

Thüm et al. [96] defined feature-based analysis as follows: "An analysis of a software product line is feature based if (a) it operates only on domain artifacts and (b) software artifacts belonging to a feature are analyzed in isolation (i.e., knowledge about valid feature combinations is not used) and feature interactions are not considered".

From this definition, it can be inferred that feature-based analysis is similar to family-based analysis in the sense that both strategies operate on domain artifacts rather than on generated products. However, unlike family-based analysis, a variability model is not required in the feature-based analysis as every feature is analyzed in isolation. Since the number of all valid feature combinations might grow exponentially, the purpose of feature-based analysis is to reduce the potentially exponential number of analysis tasks (i.e., for every valid feature combination) to a linear number of analysis tasks (i.e., for every feature), while tolerating the fact that the analysis *might* be incomplete.

In the literature, feature-based analysis is combined with product-based analysis to perform *feature-product-based type checking,* such as the work of Kolesnikov et al. [108], and Bettini et al. [120]. The main idea of these approaches is, first, to analyze features in isolation, and second, to ensure that all properties that are not feature-based checked are analyzed for each product.

In addition, feature-product-based analysis is used for *model checking*, where each feature implementation is model-checked in isolation and an interface is generated. Then, these interfaces are checked for every product to make sure that features are compatible with each other. The approach proposed by Liu et al. [121] is one such example.

In comparison with product-based analysis strategies, feature-product-based strategies reduce redundant computations. However, redundancies still occur for the analyses part that is applied at the product level. In addition, the performance of feature-product-based strategies depends on the actual analysis (model checking, type checking, theorem

proving, etc.), the number of products in a product line, and whether evolution of the product line is taken into consideration or not. This is typically not the case with analysis of model families using union models.

2.10. Research Area 3: Metamodel-Related Evolution Approaches

The literature discussed for this research area is related to the second part of my thesis, namely relaxing metamodels to support the representation of model families (discussed in Chapter 7). So, to build coherence about our approach, the reader can defer reading Sections 2.10 to 2.15 until after Chapter 6, without compromising the understandability of Chapter 3 to Chapter 5.

2.10.1 Overview

Throughout my literature review, I noticed an obvious lack of approaches that attempt to evolve/relax metamodels to support model families. In particular, to the best of my knowledge, there are no approaches that attempt to *anticipate* relaxation points in metamodels to support the representation of a model family (captured by a union model) as one of the valid metamodel's instances (see Figure 6 and Section 1.3).

In the literature, however, a well-addressed and partially solved problem is the metamodel evolution and model/constraint/transformation co-evolution problem (Figure 13). Such problem is conceptually related to the model family-specific problem discussed in Section 1.3. In fact, it could be viewed as its opposite, since the model family-specific problem results, in a general sense, from the metamodel co-evolution that is triggered by model evolution, as illustrated in Figure 14. The literature suggests attempts to manage evolution of metamodels and co-evolution of MDE artefacts in three directions:

1) Studying the impact of metamodel evolution on models and adapting models to their evolving metamodel. This direction is known as metamodel evolution and model co-evolution (illustrated in Figure 13);
2) Studying the impact of metamodel evolution on the entire ecosystem of models and operations (such as migration and transformations) and then adapting operations to

the evolved metamodels [122]. This direction is known as metamodel evolution and operation co-evolution; and

3) Studying the impact of metamodel evolution on the external constraints (e.g., OCL) and adapting the latter to the evolved metamodels.

In the three directions, the goal is to update modeling artefacts (i.e., models, operations and/or constraints) so that they conform to the evolved metamodel [123][124]. The subsequent sections discuss each one of the three directions.

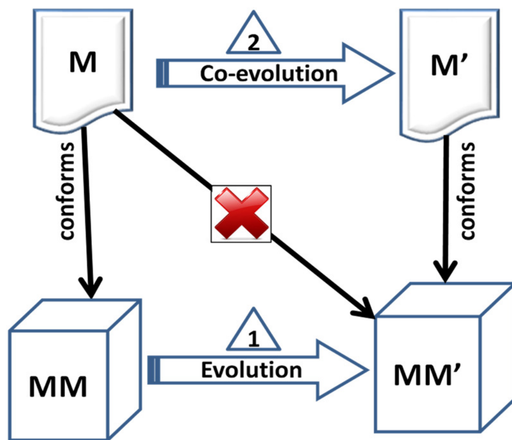


Figure 13 Metamodel (MM) evolution and model (M) co-evolution problem

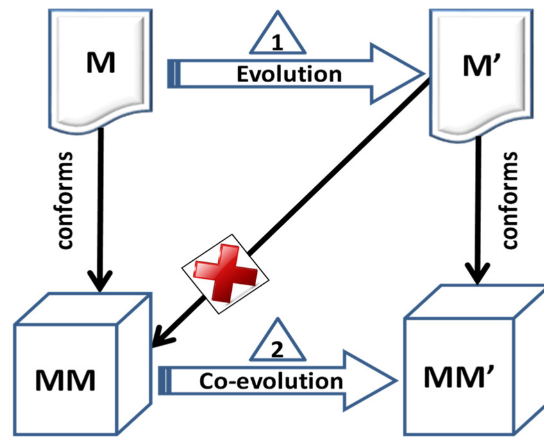


Figure 14 Model-triggered metamodel (MM) evolution problem (general)

2.10.2 Metamodel Evolution and Model Co-evolution

In general, metamodel evolution and model co-evolution approaches can be classified into three categories [125] (1) *Manual approaches*, where migration strategies are encoded manually by the modeler using general purpose programming language (e.g., Java), or transformation languages, such as the Atlas Transformation Language (ATL) [126] OMG's Query/View/Transformation (QVT) [127], (2) *Metamodel-matching techniques*, in which, migration strategies are inferred from the difference between the original metamodel and the evolved metamodel [128][129], and (3) *Operator-based approaches*, where metamodel changes are recorded as a sequence of co-evolutionary operations used later to infer a complete migration strategy [130][131][132].

To handle metamodel and model co-evolution, several approaches create difference models to calculate the differences with the last evolved metamodel [133][134]. Then, this difference model is used to derive automatic transformations for co-evolution. Sprinkle proposed a visual graph-transformation-based language [135][136] in order to specify model migration between two metamodel versions. In this approach, co-evolution of models is tackled by designing co-evolution transformations based on metamodel change types. Becker et al. [137] and Gruschko et al. [138] classify primitive metamodel changes into non-breaking, resolvable, and irresolvable changes. Then, they provide automatic migration rules for non-breaking and resolvable changes. Cicchetti et al. [139] go a step further and try to detect composite changes (e.g., extracting a class based on the difference between metamodel versions). The authors compute differences between two metamodel versions that are then taken as input to adapt models automatically. This is achieved by transforming the differences into a migration transformation with a so-called higher-order transformation. In order to adapt to metamodel changes and migrate models, a number of high-level transformations were proposed in [124]. These transformations are based on a generic model that supports versioning for both models and metamodels.

Wachsmuth [131] adapted ideas from object-oriented refactoring and grammar to provide the basis for metamodel evolution and model co-evolution and to present other types of changes. In [140]-[143], the authors proposed co-evolution approaches that base their solution on "in-place" transformations (i.e., transformations that update an input model to produce the output model). In such approaches, the co-evolution rules are specified as in-place transformation rules by using a kind of unified metamodel representing both metamodel versions. Finally, search-based approaches have been used in model-driven engineering [144]. In particular, they were used for metamodel evolution and model co-evolution. Williams et al. [145] proposed the use of search-based algorithms to reason about possible model changes using metamodel differences computed before the coevolution of models can take place. In the same context, Kessentini et al. [146] used a different approach to address the problem of co-evolution of models without the need for computing differences on the metamodel level. The authors viewed the co-evolution process as a multi-objective optimization problem and solved it using the Non-dominated Sorting Genetic Algorithm II (NSGAII). In particular, the algorithm works on finding the best

operation sequence that generates, from the initial model, a new model version conforming as much as possible to the evolved metamodel.

In contrast to our Anti-MeReFam method, none of the above approaches investigates adapting metamodels in response to the evolution of models.

2.10.3 Metamodel Evolution and Transformation Co-evolution

The co-evolution of transformation in response to metamodel evolution has been recently studied in the literature, either on its own [147] or coupled with models [148]. Levendovszky et al. [149] propose a semi-automatic approach that is able to automate certain parts of the evolution. When automation is not possible, the algorithm alerts users about the missing information, which can then be provided interactively after the automatic part of the interpreter evolution. Mendez et al. [150] discuss the problem of metamodel evolution and transformation co-evolution and view the relation between metamodels and transformations as a domain conformance issue. The authors propose an adaptation process that includes three phases: impact detection, impact analysis, and transformation adaptation. Gracia et al. [151] propose an approach for transformation co-evolution in two main stages: the detection stage, where changes to the metamodel are detected and classified, and the co-evolution stage, where the required actions for each type of change are performed.

Di Ruscio et al. [152] present an approach for the coupled evolution of metamodels and transformations. This approach tries to assess the cost of a change and use this assessment to infer transformation evolution (for instance, whether to make a go/no-go decision to evolve transformations or not). Roser et al. [153] propose an approach to enable semi-automatic transformation migration after metamodel evolution. This approach takes into consideration scenarios where the source and target metamodels belong to the same knowledge area represented by a reference ontology.

Although all of the above approaches involve operation co-evolution triggered by a metamodel evolution (where the evolution of a metamodel is tracked through maintaining a difference model), the idea of these approaches is still conceptually different from our investigated problem/approach, which aims to co-evolve metamodels triggered by model's evolution/variation, current or predicted, in the context of model families.

2.10.4 Metamodel Evolution and OCL Co-evolution

The metamodel evolution and OCL co-evolution approaches that exist in the literature can be classified into two categories: *offline* approaches and *online* approaches. Offline approaches perform co-evolution of the OCL constraints after the metamodel has been evolved completely. In online approaches, on the other hand, an instant co-evolution for each change is performed during the metamodel evolution. With respect to automation, both categories are either semi-automated or fully automated.

For offline approaches, Cabot et al. [154] focused on metamodel changes that involve deleting elements. Particularly, they target the deletion of the parts of OCL constraints that use the deleted elements. Kusel et al. [148][155] then propose resolution actions in model transformation by means of ATL helpers through analyzing the impacts of metamodel evolution on OCL. Khelladi et al. [156] propose a semi-automatic approach that records changes to the metamodel in chronological order. After that, high-level changes are detected in order to apply necessary resolutions to adapt OCL constraints based on the structure of the impacted OCL constraint and the impacted location.

Online approaches outperform offline approaches in the sense that no hidden changes are missed, and the order of changes is preserved. Examples of online approaches include the approach proposed by Demuth et al. [157][158]. In this approach, the authors define a fixed structure for OCL constraints as templates that are then instantiated to update the constraints. Marković et al. [159] propose a QVT-based approach, in which they formalize refactoring rules for class diagrams and classify them with respect to their impact on OCL constraints. Similarly, Hassam et al. [160] propose a semiautomatic approach, using the QVT transformation language, to formalize the adaptation that should be applied on impacted constraints after each operation on a metamodel.

With respect to automation, the approaches of Hassam et al. [160], Khelladi et al. [156], and Kusel et al. [148] are semi-automated, whereas the approaches of Marković et al. [159], Cabot et al. [154], and Demuth et al. [157][158] are fully automated.

The above-mentioned approaches focus on identifying high-level changes to the metamodel so as to co-evolve OCL constraints accordingly. Conversely, our metamodel relaxation method aims to identify changes on the model-level and then co-evolve metamodels thereafter.

2.11. Metamodel Extension Approaches

Metamodels can be extended through the concept of *profiles*. A well-known example is the UML.2x profile mechanism [161], which exploits three concepts: stereotypes, tagged values, and constraints. Although profiles are used as metamodel extension mechanisms, they should not be perceived as being able to remove or change existing definitions of a metamodel. Rather, they provide a "back-door" mechanism through which new constructs (stereotypes) and properties (tagged values) can be added. In addition, modeling rules (constraints) are added via profiles to further *restrict* the metamodel's constructs and enforce the well-formedness of models of the domain-specific language. Similarly, Ecore, the metamodeling language of EMF, allows users to attach annotations on any element of a metamodel to capture additional information [162]. In addition, approaches for (meta)model decoration/annotation, such as the one from Kolovos et al. [163], have also been used in an extension context, to represent usage-specific information.

Unlike the proposed metamodel relaxation methods, these approaches either add new concepts to the original metamodel or modify the language's validity constraints by further *constraining* them instead of *relaxing* their restrictions.

2.12. (Meta)Model Versioning Approaches

Versioning or differencing approaches compute the so-called "*difference model*" between two (meta)model versions. The difference models usually contain atomic changes such as added, deleted, and updated elements, and some complex changes, such as property moves.

Several model versioning approaches have been proposed in the literature. The approach of Alanen and Porres [164] is one of the earliest works on UML model versioning. In this approach, differences between versions of the same model are detected by calculating the created, deleted, and changed elements, and then by matching the unique identifiers of these elements. Odyssey-VCS 2 is a version control system for UML models [165]. It controls versioning by using state-based differencing to detect elements between different versions of a model. The Adaptable Model Versioning (AMOR) is another model version control system proposed by Altmanninger et al. [63], which provides a mechanism for conflict detection between models by supporting definitions of conflict resolution policies. In

addition, AMOR contains a recommender component that provides suggestions to users on how to resolve the detected conflicts. A model repository for EMF models, called EMFStore, was proposed by Koegel et al. [166]. EMFStore enables model versioning by tracking all modifications undergone by a model. The modifications are then checked out and committed to the repository so as to update the last versions of a model. Finally, Mussbacher [94] explicitly extended the metamodel of GRL to document explicit changes (additions/deletions) of model elements to specific versions of a metamodel. Although a model family can then be captured, this approach is specific to one language and currently incomplete in the kinds of changes to versions it can accommodate.

Although these approaches handle model evolution and track it through versioning, none of these approaches exploits model evolution as a trigger to evolve/extend the original metamodels to enable conformance.

2.13. Metamodel Relaxation Approaches

Different approaches in the literature tend to relax the constraints imposed by a language metamodel to facilitate/perform a particular task. For instance, Syriani et al. [167] proposed an approach to semi-automatically generate a domain-specific pattern language from an input (or output) metamodel through a process called *RAMification*. RAMification is the process of modifying a metamodel of a language in a particular way so that it can be used in the patterns of transformation rules. The RAMification process is specific to languages for which the metamodel is defined by a UML class diagram and it involves three adaptation phases: *Relaxation*, *Augmentation* and *Modification*, hence, the name RAMification. The relaxation step relaxes *all* constraints of lower multiplicities of every association end imposed by the metamodel (e.g., a 1..2 multiplicity is relaxed to 0..2). In addition, it allows the instantiation of classes that were originally abstract, i.e., it makes all abstract classes concrete. The augmentation step involves augmenting element of the relaxed metamodel with additional information, constraints and attributes to be used by the transformation engines (such as labels of pattern elements and parameter passing between different rules). Finally, the modification step modifies the data type of all attributes to "constraint" so that they can express constraints on attribute values or "actions" that compute the new value of the attribute.

Ramos et al. [168] proposed an approach to generate and represent model patterns (called model snippets) in order to use them for pattern matching. However, patterns are expressed in a higher level of abstraction than the models themselves, hence, the original metamodel of the intended models to be matched is too restrictive to represent patterns. To solve this issue, Ramos et al. construct a more flexible metamodel (i.e., a relaxed metamodel) that allows the representation of abstract patterns with all concepts of the original metamodel. This relaxed metamodel MM' is obtained by removing all restrictions that exist in the original metamodel, MM, such that: (1) no invariant or pre-condition is defined in MM', (2) MM' has no abstract elements and (3) all features of all classes in MM' are optional. To allow features to be optional, their lower bounds are set to zero.

Kramer [169] proposed an approach for model weaving where a user is required to be able to formulate pointcut and advice model snippets for the metamodel of their choice. These model snippets, however, do not need to satisfy all constraints defined in the original metamodel, MM. Therefore, two metamodel variants with relaxed constraints were obtained to define pointcut and advice model snippets. In these metamodel variants, constraints (such as lower bounds and abstract classes) are relaxed in order to allow the definition of incomplete model snippets. The use of relaxed metamodels makes it possible to define aspects with pointcut models that specify only the part of a model region that is relevant for the application of the aspect. Following the approach of Ramos [168], the relaxed metamodel variants, MM', are obtained from MM such that pre-conditions and invariants are removed, all features of all metaclasses are specified as optional (i.e., lower bounds for references are set to zero), and all abstract metaclasses are made concrete (i.e., direct instances are no longer forbidden).

Sen et al. [170] proposed an integrated methodology and semi-automated tool for testing model transformations using partial models, where users would be able to build only small partial test models directly representing their testing intent. In order to represent these partial models, the authors proposed to have a relaxed version of the original metamodel of the model transformation. The purpose of using a relaxed metamodel in this context is to enable the specification of elements in a modeling language without obligatory references, containments, or general constraints that were in the original metamodel. As in

Kramer's work [169], Sen et al. [170] adopted the transformations proposed by Ramos et al. [168] to generate a relaxed metamodel suitable to specify a partial model.

Another type of metamodel relaxation approach, known as *metamodel pruning*, is proposed in the literature. The basic idea of metamodel pruning [171][172] is to take a metamodel as input and derive a simplified/pruned metamodel as an output by removing some classes, references and/or attributes [173]. When a class is pruned from a metamodel, all subclasses, all attributes and incoming or outgoing references need to be also removed. In addition, when removing certain metamodel elements by pruning, related structural constraints (such as multiplicity, inverse, etc.) should be removed so as to obtain a consistent pruned metamodel.

In each of the above approaches, a relaxed (or pruned) metamodel was obtained to enable the description of incomplete models (which can often be encountered in case of design patterns) and/or to facilitate the representation of partial models or model snippets that do not need to satisfy all constraints defined in the original metamodel. Such relaxed metamodels are obtained based on MM by removing *all restrictions* that exist in MM including invariants and mandatory features, and by enforcing the nonexistence of instances of certain class. Our proposed approach is different from these approaches in the sense that it does not target the relaxation of all MM constraints. It aims to relax only *particular* constraints related to multiplicities of attributes and association ends. In addition, the context and the purpose of relaxing metamodels in such approaches is different than the purpose of our relaxation method, which is mainly to support the representation of model families.

2.14. Uncertainty Management Approaches

To manage uncertainty during the software development lifecycle, many approaches propose the use of *partial models* to represent and express partial knowledge and postpone decisions. For example, to manage design-time uncertainty in MDE, Famelis et al. [174][175] proposed the use of *partial models* as formal development artifacts to enable the abstraction, visualization reasoning and manipulation of possible alternative design models. In this approach, a set of alternative models with uncertainty are merged and captured with one model called partial model. To allow the representation of partial models,

the authors proposed a process called *metamodel partialization*, such that given a base metamodel, M_B, a corresponding partial metamodel, M_P (called *partialized metamodel*) is derived. M_P contains the same meta-elements as M_B, but with the following adaptations: 1) every meta-element has an additional Boolean meta-attribute (called *isMaybe*), 2) M_p contains an additional singleton meta-class (called *MayFormula*) that has a single attribute formula, and 3) the multiplicity constraints of every meta-association are *unbounded* (typically denoted by “*").

To manage uncertainty in bidirectional model transformations, Eramo et al. [176] propose the use of an *Uncertainty Metamodel*, U(M), to reduce the burden of managing a collection of models. The U(M) is obtained by *extending* a base metamodel M with specific uncertainty points to represent the multiple outcomes of a transformation. In this approach, the base metamodel, M, is extended by adding more abstract subclasses, generalizing some meta-classes, adding direct-subclasses and relaxing *all* the cardinality of attributes and references and make them optional so as to permit the expression of uncertainty.

Our proposed approach is different from Famelis' approach [175] in two major aspects: while the idea of capturing models in one partial model is close to our idea of merging models of a family in one union model, the context and the purpose of our work is still different. In a sense, we do the merge of models for the sake of representing and analyzing model families, and then we relax metamodels (at particular relaxation-points) accordingly. In [175], however, merging models is done to describe the observable behaviour of a system. In addition, the work of [175][176] proposes to relax *all* multiplicity constraints related to association ends and attributes, an approach that is different from our method, which aims to relax particular constraints only, in a minimal way, in order to minimize the adaptations required to analysis techniques and editing tools.

2.15. Database Schema Evolution Approaches

Database schema evolution has been a field of study for several decades, yielding a substantial body of research [177][178]. These approaches deal with the migration of database records to adapt to the evolution of database schema [179]. A variety of model evolution approaches have fully adopted the key ideas from database schema evolution or got inspired by them. Details about schema evolution approaches are beyond the scope of this

thesis, but the interested readers can refer to the work of Rahm [180] and Roddick [181]. While many of the approaches from schema evolution have been adapted for model evolution, one key consideration has yet to be fully addressed: if a database record evolves, or a new record is added that is not already defined in the database schema, then there is a need to evolve the schema to insure compatibility with the new record. This issue is more or less analogous to the model-triggered metamodel evolution problem (Figure 14), investigated in Chapter 1.

2.16. Chapter Summary

This chapter surveyed research areas potentially related to the main themes of this thesis. Section 2.1 discussed the review methodology. Sections 2.2 to 2.6 discussed variability management and modeling in MDE. Section 2.7 and Section 2.8 discussed variability management and modeling using union model-like artifacts and goal models, respectively. Analysis approaches that target groups of models are discussed in Section 2.9. Finally, Sections 2.10 to 2.15 reported on metamodel-related evolution approaches. The chapter not only discussed the state-of-the art literature, but also identified the gaps that exist in the current approaches and illustrated how this thesis' work is different. These differences can be summarized as follows:

- The concept of union models to represent and capture model families has not been discussed in the literature.
- The annotation mechanisms used in the literature are different from our proposed annotations. To the best of our knowledge, the spatio-temporal annotation language that we use to annotate elements of union models with information about versions/variations has never been proposed in the literature.
- None of the approaches discussed in Sections 2.3 to 2.6 consider the evolution of models over time and the variation over the space dimension in the way that we handle it (i.e., using one single, annotated union model). In particular, SPL-related approaches focus on modeling variability over the space/product dimension. VC-related approaches focus on handling and managing variability of models over the time dimension. Even the approaches that handle variability in

space and time together (Section 2.6) address the problem from the perspective of SPL, which is different from our approach.

- Unlike the use of union models (which does not require feature models), all SPL-related approaches (Section 2.3 and Section 2.5) rely heavily on the existence of feature models to construct 150% models and to extract individual models from them.
- The usage of union model-like artifacts (mainly the 150% models), as discussed in Section 2.7, does not go beyond the ability to extract products to conduct only particular and traditional types of analysis (such as testing, model checking, typed checking, or theorem proving). However, our method aims to exploit union models (with annotations) to perform more sophisticated kinds of analysis, such as trend analysis.
- The use of goal models to manage variability (as discussed in Section 2.8) did not consider variability of models along the time dimension.

At the metamodel level, this chapter revealed a lack of approaches that deal with metamodel co-evolution in light of model evolution in the general context (Figure 14) and in the context of model families in particular (Figure 6), which is the typical problem discussed in this thesis. The gaps between the existing research and our proposed research are discussed in Sections 2.10 to 2.15 and summarized as follows:

- Most approaches conduct transformation/migration of models each time a metamodel changes (Section 2.10); this requires a non-trivial adaptation effort for tools and analysis techniques. Our proposed approach, on the other hand, aims to infer a single relaxed metamodel that accommodates all potential union models of a language, so as to develop tools for this relaxed metamodel only *once.*
- The driving factor of evolution is different. While existing approaches deal with models and/or operations co-evolution triggered by metamodel evolution, our work targets the evolution/relaxation of metamodels triggered by model evolution in the context of model families (Section 2.10 and Section 2.12).
- Some of the related approaches (Section 2.11) either add new concepts to the original metamodel or modify the language's validity constraints by further

constraining their restrictions, whereas our approach only intends to relax some constraints (as new concepts are not required).

- All the approaches discussed in Section 2.13 and Section 2.14 propose to relax *all* constraints/restrictions that exist in a metamodel (including invariants and mandatory features, and by enforcing the nonexistence of instances of certain class). This extreme relaxation would have a negative impact on tools adaptation. Our proposed approach, on the other hand, is different in the sense that it aims to relax particular constraints related to multiplicities of attributes and association ends, together with OCL-like constraints, in a minimal way.
- Finally, the context in which a metamodel is relaxed and the purpose of relaxing metamodels in all existing approaches is completely different than the purpose of our proposed relaxation method (which is mainly to support the representation of model families, not to generate new types of products).

Chapter 3. Characterization of Model Families

This chapter provides a comprehensive characterization of the concept of *model families* by drawing an analogy with a well-investigated concept, namely software families, and discusses our proposed annotation language, STAL, suggested to support the representation of model families.

This chapter is organized as follows: Section 3.1 presents the concepts of software families and their associated variability in the space and time dimensions. These concepts are used as a basis to establish an analogy between software families and model families, as discussed in Section 3.1.3. The special characteristics of model families are discussed in Section 3.2. Section 3.3 elaborates on the challenges associated with model families at both the model level and the metamodel level, and reports on the need for generic models to represent and capture model families. Our proposed annotation language, STAL, is discussed in Section 3.4. Finally, Section 3.5 summarizes this chapter.

3.1. Model Families

This section discusses model families and explains their main characteristics. To pave the discussion about model families, this section uses an analogy between this relatively new concept and an already existing and well-investigated concept, namely *software families*[5].

3.1.1 Preliminary: Software Families

According to Pohl et al. [3], "*if a set of software systems has a significant common core functionality and is further subject to a software engineering process handling variability, one may speak of a software family*". In other words, a software family is a set of systems that consists of assets shared by all members (i.e., *commonalities*) and assets used only by a subset of members (i.e., *variabilities*). Pohl et al. [3] defined commonality as follows:

---

[5] The term "software family" is used more frequently in Europe, whereas in North America, the term "software product line" is used more often as a synonym.

"*Commonality denotes features that are part of each application in exactly the same form*". In addition, they defined variability as follows: "*variability is modelled to enable the development of customised applications by reusing predefined, adjustable artefacts*". In this context, variability modeling (e.g., using feature models) is *required* to provide configuration rules, where these rules govern which combinations of variabilities (when combined with commonalities) are deemed valid.

Software families could vary in different shapes according to their state of development, functionality, or targeted platform. Different *dimensions* can be used to distinguish such variability, including 1) variability in *space* and *time* and 2) internal and external variability. This thesis only focuses on the first dimensions (i.e., variability in space and time) used for the analogy. However, the interested reader can refer to Pohl et al. [3] for more details about other variability dimensions.

3.1.2 Variability in Space and Time

In the literature, several authors (such as Pohl et al. [3], Elsner et al. [182], and Schubanz et al. [183]) discussed variability in the time and the space dimensions, as well as their interrelationships. *Variability in space* represents all those changes that alter the functionality of software, referred to as the *configuration* space. Pohl et al. [3] defined variability in space as: "*the existence of an artefact in different shapes at the same time*".

An example of software families that vary over the space dimension is Microsoft Office 2016 (shown in Figure 15) with individual products (or configurations) such as Word, Excel, and PowerPoint. These products have different intended usages of the individual applications, but also, they have common features or functionalities, such as the export to Portable Document Format (PDF) or support for different fonts [73].

In addition to the fact that software families are subject to variability in space, they could also be subject to other changes over the course of *time*. For example, a particular product (or configuration) could change over time to satisfy new emerging requirements or to fix defects in order to maintain usability and consistency of systems. The changes of software systems over time is known as *evolution,* and it results in a set of new *versions* of the affected software artifacts. Hence, software evolution can be regarded as another dimension of variability and is referred to as *variability in time*. According to Pohl et al. [3],

variability in time is defined as: "*the existence of different versions of an artefact that are valid at different times*".

Figure 16 shows an example of a model family that results from variability in time. In Microsoft Office, the evolution of an individual product (or configuration) creates multiple versions of that product, such as Word 2007, Word 2010, Word 2013, and Word 2016.

Figure 15 Microsoft Office 2016 as a software family with variability in space (i.e., configurations)

Figure 16 MS Word as a software family with variability in time (i.e., evolution)

Variability in space and variability in time could seem orthogonal and each dimension might be treated in isolation from the other. However, this is not always the case, and in fact, software artifacts vary in time as well as in space. Hence both dimensions of variability could be *interdependent*. On the one hand, the existence of certain configurations (with their dependencies regarding configurable functionality) may require further evolution. On the other hand, the evolution of an artifact over time may affect configuration options (for example, due to technological constraints). Hence, it is important to consider time as another dimension of variability and not to focus only on the space dimension.

Getting back to the example of Microsoft Office suite, the dimensions of variability in space and time are transparent to end-users since users are only concerned with individual products. Consequently, end-users may combine different products (or configurations) and different versions of these products, but they cannot combine different variable functionality of products or different versions of variable functionality [73]. For instance, a user can decide to use Word 2016 with Excel 2013, but she cannot use the spreadsheet calculation functionality of Excel 2013 (directly) within Word 2016 or the revised version of the PDF export of a later version with the base functionality of an older version. Figure 17

shows Microsoft Office as a software family that results from variability in space (i.e., configuration) and variability in time (i.e., evolution) where both dimensions can be handled simultaneously.

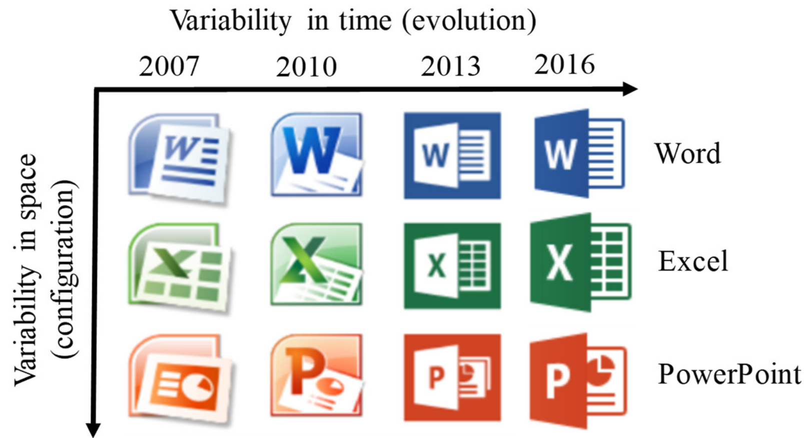


Figure 17 Microsoft Office software family with different configurations (vertical) and several versions (horizontal) [73]

3.1.3 Model Families: An Analogy with Software Families

Having discussed software families along with their dimensions of variability and given that a model family constitutes a set of models with common elements as well as different elements, we argue that a model family is conceptually similar to a software family. Hence, we define a model family (MF) as: *a set of related models with commonalities and variabilities between individual models*.

As in software families, models in a model family could be subject to changes that alter their *configurations* (e.g., in the regulatory domain, when there exist different variations of regulations targeting different types of companies that need to be modeled), or as part of models' evolution over *time*, where evolution produces a set of related model versions (e.g., when regulators need to react to new government decisions, over time).

Having said that, variability in model families, just like in software families, can be defined using the dimensions of space and time. Based on their main dimension of variability, we distinguish between two categories of model families:

- Model families that result from variability in *time*, which we refer to as *Model-Versions Family* (MvF).

- Model families that result from variability in *space*, which we refer to as *Model-Configurations Family* (McF).

Analogous to Pohl et al.'s definitions of variability in space and time, we define MvF as: *the existence of different versions of a model that are valid at different times*. In the same manner, we define McF as: *the existence of a model in different spaces (or configurations) at the same tim*e.

MvFs result from evolution of models over the course of time, where evolution produces a set of related model versions with differences and similarities between versioned models. McFs, on the other hand, can be found in model-based (software) product lines (MDPLE, see Section 2.3), where different configurations (or variants) of a product can exist simultaneously for different customers, without necessarily being caused by evolution over time. In addition, McFs could result from having several design alternatives (at the same time) due to uncertainty that exists at multiple stages of the development process [175].

Figure 18 (a) shows the concept of MvF, where a model M could evolve over time into several versions, such as 1st version (M_{v1}), 2nd version (M_{v2}), 3rd version 3 (M_{v3}), etc. The concept of McF is also shown in Figure 18 (b), where a model M has different configurations: configuration 1 (M_{c1}), configuration 2 (M_{c2}), configuration 3 (M_{c3}), etc., which all exist at the same time (i.e., Jan 2019).

Considering both dimensions of variability and following the rationale illustrated in Figure 17 (where software families vary in time as well as in space), model families can be *"ideally"* represented as a set of inter-related MvFs and McFs, as shown in Figure 19. The reason we say "ideally" here is because the representation of model families in this way does not always reflect the real world. Actually, we should not ignore the fact that the evolution of model configurations does not happen simultaneously all the time. In other words, it is not mandatory that, e.g., the 1st version of configuration 1 of a model M (i.e., $M_{c1, v1}$) will evolve into $M_{c1, v2}$ with the same timeline as $M_{c2, v2}$. In addition, it is not necessary to have an equal number of configurations for each model version. That is, we could have, e.g., three configurations for the 1st version of a model (i.e., $M_{c1, v1}$, $M_{c2, v1}$, $M_{c3, v1}$) but only two configurations for the 2nd version ($M_{c1, v2}$, $M_{c3, v2}$) with one configuration

missing for any reason. Having said that, a more *realistic* characterization of a model family would look like the one illustrated in Figure 20, with some missing versions or configurations to indicate the absence of some evolution over the time dimension or variation over the space dimension, respectively. It is worthwhile to mention that the same model can be part of multiple families, e.g., a model M could be a part of an MvF and an McF.

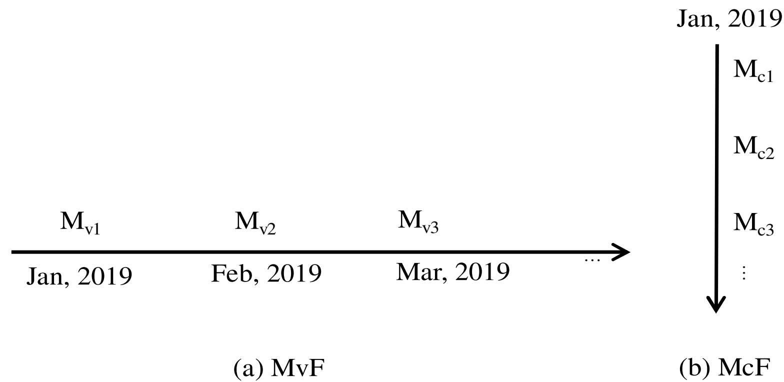


Figure 18 (a) MvF with several versions over time, (b) McF with several configurations

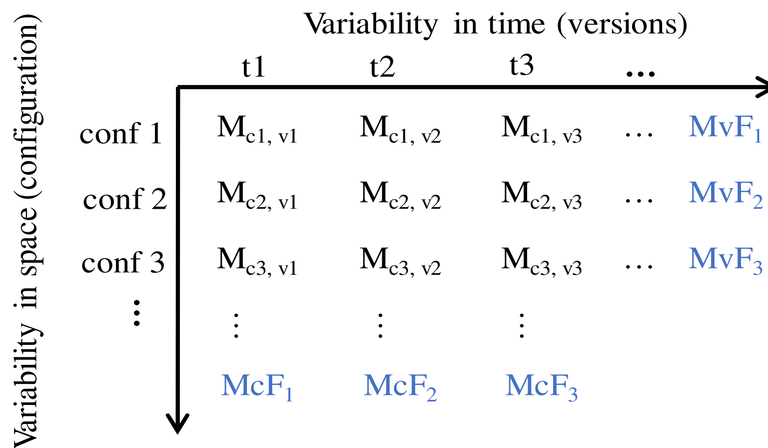


Figure 19 Model families in both space and time dimensions

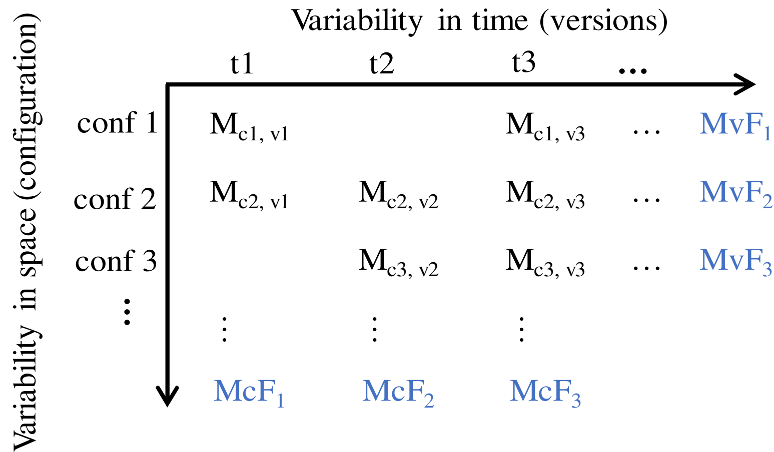


Figure 20 A more realistic model families in both space and time dimensions

3.2. Special Characteristics of Model Families

Besides the characterization provided in Section 3.1.3, model families have also the following special characteristics:

1) For a given language, each individual model in a model family conforms to the same metamodel since these models are expressed using the same language (e.g., UML, or GRL). That is, models in a model family are *homogenous* (see Figure 21). In this thesis, we exclude those *heterogenous* models that conform to different metamodels (i.e., a group of models from different languages), as shown in Figure 22.
2) The evolution/variation of models in a family does not involve any additions, deletions, or modifications to language concepts. Hence, the metamodel to which family members conform need not to be extended or altered. Rather, variability in model families might only affect some of the internal constraints that are related to multiplicities of attributes and association ends and/or some external well-formedness constraints, as we discussed in [185][186]. See also Chapter 6 for more details.

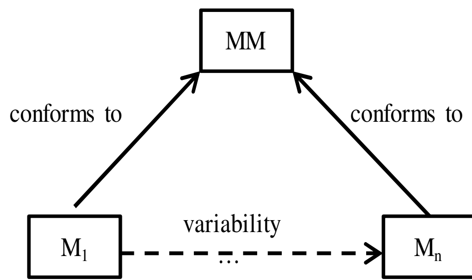


Figure 21 Homogeneous models (a model family)

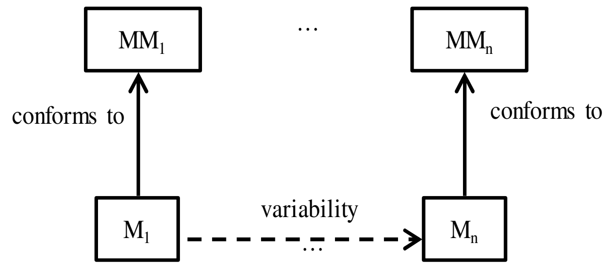


Figure 22 Heterogeneous models (out of thesis scope)

3.3. Challenges of Model Families

This section discusses the challenges associated with model families, where some of these challenges are also witnessed in software families, while others are particularly specific to model families due to the latter's nature.

3.3.1 Variability-related Challenges at the Model Level

The existence of several versions of models over time (i.e., MvF) or various configurations of a model at the same time (i.e., McF) is a non-trivial issue. In fact, model families are subject to frequent evolutions over time (e.g., to adapt to new or changed requirements) that are asynchronous by nature. The unsynchronized evolution, sometimes, requires older versions of models (which represent legacy systems) to be maintained as they may still be in use even after they were superseded by newer versions. For example, the evolution of a model M_{v1} in Figure 18 creates two more versions M_{v2}, M_{v3} of that model, where it is possible for these versions to be used and co-exist together. The challenge in this context is that the *analysis* of each model version, *individually*, becomes impractical, especially if the number of models is large, or if the evolution frequency is high. Intuitively, if there are *M* individual models in a model family, and each model has *E* elements, then the complexity of analyzing all models, *one at a time*, would be in order of $O(M \times E)$. Such complexity becomes more significant if there are hundreds of models, with hundreds of elements in each model. It is worthwhile to mention that even though software families face principally similar issues, evolution over time is often treated as a separate phenomenon, where a software family advances to a new state without preserving previous versions of artifacts [184].

The above-mentioned issue is encountered over the time dimension and can also be witnessed in the space dimension. In particular, model *configurations* are not necessarily known in their entirety, leading to a completely open configuration space where new configurations may appear, and previously existing ones may vanish independently. For example, if we consider modeling regulatory domains, a regulator may impose new regulations or decide to discontinue support of some regulations in between two releases of regulations. In that case, if we need to analyze all models together, a mechanism is needed to keep track of all configurations.

A yet more challenging situation is when both dimensions of variability of model families may affect each other such that they cannot be handled in isolation for all families. For instance, evolution of models could modify some available configuration options or rules due to new constraints imposed by economy or technology. In addition, the existence of certain configurations of models, with some configurations being interdependent, may require further evolution of models. In this case, managing variability requires considering both dimensions of variability without neglecting any one of them or treating them separately.

3.3.2 The Need for Generic Models to Represent Model Families

Having discussed the challenges associated with model families, a mechanism is needed to capture the entirety of a model family (in *both dimensions* of variability), in a comprehensive and exact way such that all and only individual members of a family can be modeled or analyzed. Intuitively, if there is a set M of individual models, each model with E elements, then the complexity of analyzing the set M of individual models, *all at once*, using a single generic model would be of order $O(E')+T$, where T is the time of constructing the generic model (which could be negligible in some scenarios), and E' is the number of elements in that generic model which captures all family members. It is worthwhile to state that E' is usually greater than (by some ratio) or equal to E (introduced in Section 3.3.1). In the worst case, when all individual models are distinct (i.e., have no elements in common), E' could be equal to the summation of all elements in all individual models. However, this is an extreme case where, if it happens, the use of the called-for generic models to capture model families would not be recommended.

In this thesis, we propose a *union model (M_U)* as a generic modelling artifact to capture all model elements in all model members of a model family MF, aggregated in a compact and exact way that enables the extraction of each individual model of the family. Generally speaking, a union model M_U of any potential model family (i.e., an McF or MvF in Figure 19) would be the union of all elements, e, in all individual models of that family, MF. That is:

$$M_U = \bigcup_{i=0}^{n} e_i \in Mi \, , \forall \, Mi \in MF \qquad (1)$$

A very ***important requirement*** in M_U is to be able to represent and to distinguish variability among elements of member models, i.e., to which model(s) a particular element belongs, and to enable the extraction and the analysis of specific members. To achieve this, we propose a *spatio-temporal annotation language (STAL)* to annotate elements of each individual model with information about their versions and/or configurations. The details about STAL are discussed in Section 3.4.

3.3.3 Conformance-related Challenges at the Metamodel Level

Constructing a union model to capture all (and only) elements of individual models in a model family and adapting existing analysis techniques (that are usually applicable on single models) to be used on union models are themselves challenging issues that are witnessed at the *model level*. Nevertheless, while constructing the union model, conformance issues between that union model and the modeling languages' metamodel might emerge. This is because, sometimes, the modeling language may not permit to capture all variations (i.e., configurations or versions) with one model. For example, let us consider using the Goal-oriented Requirement Language – GRL [6][7] as a modeling language. GRL limits the number of links between a pair of goals to 1, whereas the union model may need many. This problem was illustrated already in Figure 5 (Section 1.3), where integrating two simple GRL models leads to a union model that is not a valid GRL model (i.e., not conforming to the GRL metamodel) as it violates the constraint on the maximum number of links between a pair of goals. A solution for this problem is investigated in Chapter 6 and was published in [185][186].

3.4. Spatio-Temporal Annotation Language (STAL)

With an appropriate formalization, the challenging part of constructing a union model is not necessarily in the union operation itself, as will be discussed in Chapter 4. Rather, the challenge is in being able to distinguish to which models a particular element belongs. For example, in the goal model family shown in Figure 23 (reproduced here from Figure 2 for convenience), we need to distinguish that *Fence rule 3* belongs only to aerodromes of TYPE 1 (i.e., configuration 1), *Fence rule 2* belongs only to TYPE 2, and so on. In addition, if there are common elements among all models (such as *ACS rule 2*), we need to indicate this in M_U in a precise and exploitable manner.

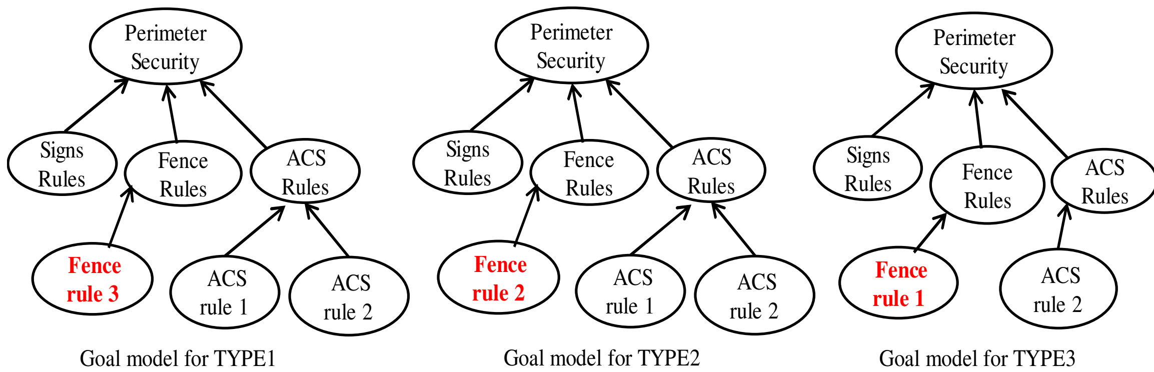


Figure 23 Goal model family of regulations (from Figure 2)

To address this issue, STAL is used to annotate elements of each individual model with space/time information in the form of $<ver_{num}, conf_{info}>$, where ver_{num} denotes the version number of a particular model (e.g., 1st version, 2nd version, and so on), while $conf_{info}$ denotes space dimension-related information (e.g., organization type, size, location etc.).

3.4.1 Syntax and Semantics of STAL

In the time dimension, models can evolve independently and asynchronously over distinct *timepoints*. Since timepoints can be corelated and compared, they naturally form a chronological order. Given this inherent chronological nature of models' evolution, a sequence of versions of a particular model can be annotated with sequential version numbers: $ver_1, ver_2, \ldots ver_n$. This creates an implicit *temporal validity* between model versions. For

instance, we can say that ver_1 happened before ver_2. The timing information embedded in the <ver_{num}> format in STAL can represent version numbers or dates, a hierarchical version numbering scheme (e.g., versions 2.3.1 and 4.3.2, etc.), or ranges thereof.

The space dimension, on the other hand, is different and somewhat more complex. This stems from the fact that the space dimension is flat and has neither a chronological order nor a hierarchical nature (except in very specific domains, such as in provinces and their cities). In STAL, we usually use the naming conventions $conf_A, conf_B, \ldots, conf_Z$ (instead of $conf_1, conf_2, \ldots, conf_n$) to reflect the lack of ordering semantics. If a configuration is simple, we use its syntactical description as a name for that configuration. For example, $conf_A$="airports in Ontario" and $conf_B$="airports in Quebec" are the names of the two different configurations of airports.

However, it is worth mentioning that information about configurations could be composite, i.e., consisting of several pieces of information. For example, TYPE1 aerodromes may refer to those airports that are of medium-size, located in Ontario, and with national flights only. To represent this type of composite information in STAL, in a way that keeps annotated models as simple as possible, we propose the use of *look-up tables*, which provide mappings between configuration names and their real descriptions. Please note that in this example, the numbering suffixes of TYPEs do not hold any ordering. meanings and they are just descriptions of the configuration. Table 1 shows an example of a look-up table for the configurations given in Figure 23.

Table 1 Mapping configurations to their detailed descriptions

Config	**Description**
TYPE1	Size=M, Location=Ontario, Flight=national
TYPE 2	Size=L, Location=Ontario, Flight=international
TYPE 3	Size=M, Location=Quebec, Flight=national

In addition, in a model family, it is possible that one model element belongs to several or all family members. For instance, assume that there is a model family with one model configuration ($conf_A$) that evolves into five versions (i.e., ver_1 to ver_5). Assume also a node *n* that belongs to the five versions of that model. In this case, *n* will be typically annotated

in the union model with five annotations: <ver_1, $conf_A$>, <ver_2, $conf_A$>, <ver_3, $conf_A$>, <ver_4, $conf_A$>, <ver_5, $conf_A$>. Such style may lead to large amounts of annotations per element. To simplify annotations of union models, the representation of STAL annotations can be simplified such that a sequence of version annotations can be represented as a range of values *([start:end])*. In the above example, the annotation of *n* becomes <[ver_1: ver_5], $conf_A$>. Ranges are however unavailable for configurations as they are usually not sortable.

In the same example, it could also happen that an element, say edge *e*, appears in all versions from ver_1 to ver_8, except in ver_4. In this scenario, a set of ranges can be used, such that *e* is annotated as: <{[ver_1: ver_3],[ver_5:ver_8]}, $conf_A$>. Furthermore, if an element *x* belongs to ver_1 to ver_3 of $conf_A$ and to versions v_1 and v_4 of confB, then such element will be annotated as <[ver_1:ver_3], ($conf_A$, $conf_B$)>; <(ver_4), ($conf_B$)>. Finally, if an element belongs to all versions and/or all configurations of a family, we annotate it with the keyword *ALL.*

The ranging mechanism used with versions may also be applicable to configurations, *only* if the latter's nature allows for continuous or discrete ranging (such as TYPE1, TYPE2, etc.). However, to avoid confusion between versions and configurations, we do not use ranges to annotate configurations in this thesis. Rather, we use a comma separated list to indicate a set of configurations, e.g., <($conf_A$, $conf_B$, $conf_D$)>. Further annotation scenarios and the full syntax of STAL are defined in Appendix B.

3.4.2 Examples

This section provides two examples of STAL annotations. The first example illustrates annotations placed on elements of a union model whose members vary in the space dimension only. The second example illustrates more sophisticated annotations, where both dimensions of variability are represented in the union model.

Example 1. Figure 24 shows a union model of the regulatory model family depicted in Figure 23, and illustrates STAL annotations placed on that model's elements. As the figure shows, some of the elements are annotated with <*ALL*> to indicate that they belong to the three of the models in Figure 23. Other elements are annotated with <TYPE1>, <TYPE2>, or <TYPE3> to show that such elements belong to configuration 1, 2, or 3, respectively.

Finally, two elements are annotated with <(TYPE1, TYPE2)> to express that these elements are part of both models of configuration 1, and configuration 2.

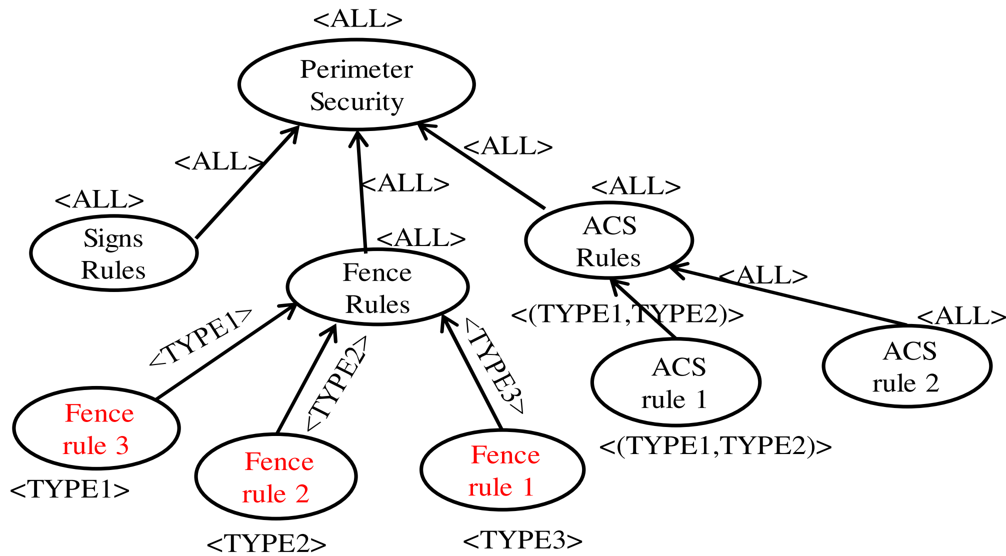


Figure 24 Annotated union model of the model family in Figure 23

Example 2. This example considers the goal model family of regulations in Figure 3, where models change over space and time, and shows how the family's union model can be annotated with STAL, with both dimensions of variability. For illustration purposes, the example considers a subset of the goal model family, namely with those models that evolve over times t1 and t2, and also over configurations TYPE1 and TYPE2. The union model that captures these four models is illustrated in Figure 25.

As in Example 1, some of the elements here are also annotated with *<ALL>* to represent the fact that they are part of all model versions (i.e., t1 and t2) and configurations (i.e., TYPE1 and TYPE2). So, in this example, the *<ALL>* annotation is equivalent to <[t1:t2], [TYPE1,TYPE2]>. On the other hand, elements annotated with <t1, TYPE1> means that they belong to the first version of the model at t1, and at configuration TYPE1. The same principle applies to other elements.

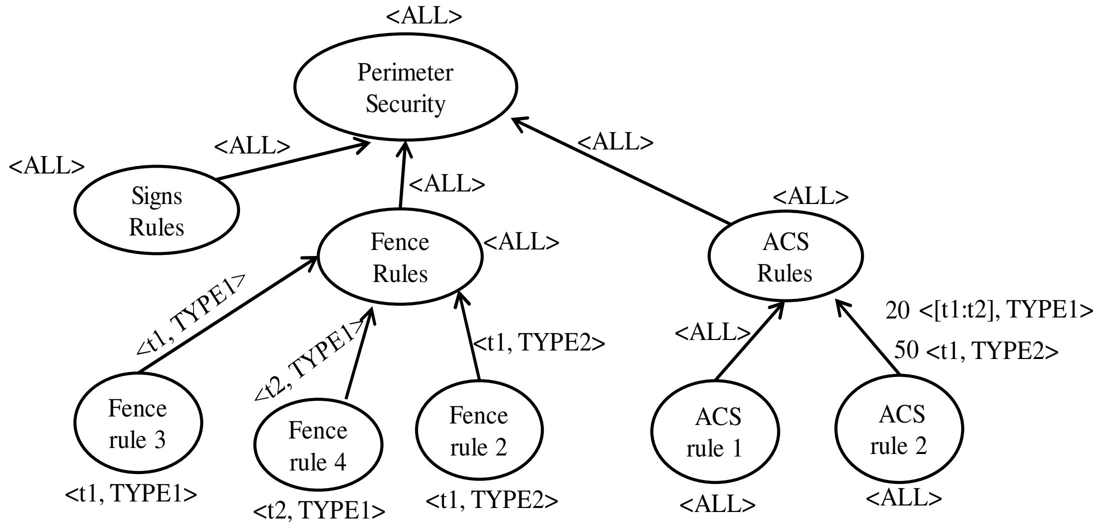


Figure 25 Annotated union model of the model family in Figure 3

It is worth mentioning here that the goal "Perimeter Security" is annotated with *<ALL>* even though that the goal model at time t2 and configuration TYPE2 is missing from the family (see Figure 3). In real world, the absence of a particular model at a specific period of time and/or configuration could be due to several reasons, one of them is that the model does not change from its previous version. In this thesis, it is assumed that a member model at time t2 and configuration TYPE2 is the same as the one at time t1 and configuration TYPE2.

3.4.3 Merging of STAL Expressions

As this thesis aims to support the union of models whose elements are annotated with STAL expressions, there is a need for a mechanism to merge STAL expressions as well. However, as mentioned in Section 3.4.1, the amount of STAL annotations per element could grow fast, depending on the number of model variations to be captured by a union model. This issue becomes more noticeable if we want to merge two union models, say M_{U1} and M_{U2}, into a new union model, M_{Unew}. In this case, annotations on each element of M_{Unew} might be complex and sophisticated. For this reason, we suggest a mechanism to unify or merge STAL expressions, in a way that the final representation of annotations would be a simplified expression (exploiting the range and list capabilities of STAL while avoiding redundancy), simpler than the simple combined listing of initial annotations.

To realize this mechanism,, annotations over space and time could be represented as binary relations in an adjacency matrix, with the matrix dimensions being time and space, and the content of the matrix being 0 to indicate no annotation for that particular time/space, and 1 to indicate the existence of an annotation for that time/space. constructing a STAL expression from a binary matrix then corresponds to a coverage of the binary relations in that matrix by a minimal number of non-enlargeable rectangles, as proposed in Jaoua et al. [187][188]. Their algorithm (called *BinaryRectangularCoverage* here) returns the minimum number of non-enlargeable rectangles (i.e., which cannot be larger due to the absence of required 1's) in a binary matrix, in a format that fits very well the format of STAL expressions, except for version ranges that need to be identified afterwards.

For example, let *Exp1* be the binary relation given in Figure 26 relating five configurations to four versions. The invocation of the *BinaryRectangularCoverage* algorithm on that matrix produces three non-elnargeable rectangles covering *Exp1*.

R1 = {Ver1, Ver2, Ver3} x {ConfA, ConfB} *// The red rectangle*

R2 = {Ver3} x {ConfA, ConfB, ConfC, ConfD} *// The purple rectangle*

R3 = {Ver2, Ver4} x {ConfA, ConfE} *// The rectangle produced from the four blue circles*

Exp1 = R1 ∪ R2 ∪ R3

Exp1	**ConfA**	**ConfB**	**ConfC**	**ConfD**	**ConfE**
Ver1	1	1	0	0	0
Ver2	1	1	0	0	1
Ver3	1	1	1	1	0
Ver4	1	0	0	0	1

Figure 26 Binary relation for expression Exp1

The corresponding STAL expression (after detecting potential ranges of versions), which can be produced by a simple transformation *ConvertRectanglesToSTAL*, is:

<[Ver1, Ver3], (ConfA, ConfB)>;

<Ver3, (ConfA, ConfB, ConfC, ConfD)>;

<(Ver2, Ver4), (ConfA, ConfE)>

Let us now have a second binary relation *Exp2*, given in Figure 27, this time relating three configurations to three versions, some of which being common with *Exp1* and some being

different. The invocation of the *BinaryRectangularCoverage* algorithm on that matrix produces three non-elnargeable rectangles covering *Exp2*.

R1 = {Ver5} x {ConfA, ConfC}

R2 = {Ver2, Ver4} x {ConfF}

R3 = {Ver2} x {ConfA, ConfF}

Exp2 = R1 ∪ R2 ∪ R3

Exp2	ConfA	ConfC	ConfF
Ver2	1	0	1
Ver4	0	0	1
Ver5	1	1	0

Figure 27 Binary relation for expression Exp2

The corresponding STAL expression (after detecting potential ranges of versions, none being present this time), which can be produced by a simple transformation *Convert-RectanglesToSTAL*, is:

<Ver5, (ConfA, ConfC)>;

<(Ver2, Ver4), ConfF>;

<Ver2, (ConfA, ConfF)>

In order to merge two STAL expressions, the MergeSTAL algorithm, presented in Listing 1, is used. This algorithm takes as input two expressions together with their contexts, composed of lists of versions and configurations (which might include elements not in the STAL expressions themselves). Such contexts are particularly useful to understand the meaning of the ALL expressions found in STAL. The output is a merged and simplified STAL expression and its resulting context. This algorithm invokes the *Binary-RectangularCoverage* and *ConvertRectanglesToSTAL* algorithms introduced earlier and a few others that are simple enough to be considered self-explanotary here.

```
Algorithm     MergeSTAL
Inputs:       Exp1:STAL, Exp2: STAL // Two STAL expressions
              // Context (all versions and all configurations) of each STAL expression
              Exp1_versions_context, Exp1_configs_context,
              Exp2_versions_context, Exp2_configs_context: List
Output:       Result: STAL  // STAL expression
              // Resulting context
              Result_versions_context, Result_configs_context : List
Intermediate:
              Matrix1, Matrix2, MatrixResult: BinaryMatrix
              RectangleList: List((List),List)) // List of (version list,configuration list)
              Rectangle: List,List // (version list,configuration list)
              Versions, Configurations: List // Version list and configuration list

Result_versions_context := Exp1_versions_context ∪ Exp2_versions_context
Result_configs_context := Exp1_configs_context ∪ Exp2_configs_context

// Special case for ALL, ALL
if (Exp1==ALL and Exp2==ALL and
   (Result_configs_context.size()==1 or Result_versions_context.size()==1) )
then Result := ALL
else {
  // Construct the binary matrices from STAL expressions by expanding their groups
  Matrix1 := MatrixExpand(Exp1, Exp1_versions_context, Exp1_configs_context)
  Matrix2 := MatrixExpand(Exp2, Exp2_versions_context, Exp2_configs_context)

  // Create a result matrix of the right size, filled with 0's
  MatrixResult := CreateZeroMatrix(Result_versions_context, Result_configs_context)

  // Insert the original matrices into the result matrix
  MatrixResult.AddMatrix(Matrix1).AddMatrix(Matrix2)

  // Compute the list of non-enlargeable rectangles based on Jaoua et al.'s algorithm
  RectangleList := BinaryRectangularCoverage(MatrixResult)

  foreach Rectangle in RectangleList do {
    Versions := Rectangle.getVersions()
    Versions.sort()
    if Versions == Result_versions_context
    then Versions := ALL // Special case for ALL
```

```
    else {
      // Extract version ranges from the list, if any
      Versions.replaceConsecutiveNumbersWithRange()
      // Turn into a STAL list if more than one element
      if Versions.size() > 1 then Versions=List(Versions)
    }
    Rectangle.replaceVersions(Versions)

    Configurations := Rectangle.getConfigurations()
    if Configurations == Result_configs_context
    then Configurations := ALL // Special case for ALL
    else {
      // No range for configurations (unsorted)
      // Turn into a STAL list if more than one element
      if Configurations.size() > 1 then Configurations=List(Configurations)
    }
    Rectangle.replaceConfigurations(Configurations)
  } // end foreach

  // Express the list of non-enlargeable rectangles as a STAL String
  Result=ConvertRectanglesToSTAL(RectangleList)
}

Return Result, Result_versions_context, Result_configs_context
```

Listing 1 MergeSTAL algorithm

To illustrate the use of the MergeSTAL algorithm, let us apply it to the Expr1 and Expr2 STAL expressions illustrated previously. Their binary relationships can simply be produced from STAL expressions by expanding their rectangles into a binary matrix (*MatrixExpand*). These matrices are already shown in Figure 26 and Figure 27. Merging these two matrices into a larger one with the corresponding context produces the binary matrix *MatrixResult* found in Figure 28. The invocation of *BinaryRectangularCoverage* on that matrix produces four non-elnargeable rectangles covering *MatrixResult.*

R1 = {Ver1, Ver2, Ver3} x {ConfA, ConfB}

R2 = {Ver3} x {ConfA, ConfB, ConfC, ConfD}

R3 = {Ver2, Ver4} x {ConfA, ConfE, ConfF}

R4 = {Ver3, Ver5} x {ConfA, ConfC}

MatrixResult = R1 ∪ R2 ∪ R3 ∪ R4

Result	ConfA	ConfB	ConfC	ConfD	ConfE	ConfF
Ver1	1	1	0	0	0	0
Ver2	1	1	0	0	1	1
Ver3	1	1	1	1	0	0
Ver4	1	0	0	0	1	1
Ver5	1	0	1	0	0	0

Figure 28 Binary relation for expression (Matrix)Result

The rest of the MergeSTAL algorithm checks whether these rectangles can lead to special cases (*ALL*), whether ranges can be identified within the sorted versions (which is the case for Ver1, Ver2, Ver3 in rectangle R1), and whether lists are required when more than one element (versions or configurations) is present. The resulting STAL expression Result, produced by invoking *Convert-RectanglesToSTAL* on the modified list of rectangles, is:

<[Ver1, Ver3], (ConfA, ConfB)>;
<Ver3, (ConfA, ConfB, ConfC, ConfD)>;
<(Ver2, Ver4), (ConfA, ConfE, ConfF)>;
<(Ver3, Ver5), (ConfA, ConfC)>

Finally, the resulting context output in this example is composed of two lists:

Result_versions_context = Ver1, ver2, Ver3, Ver4, Ver5
Result_configs_context = ConfA, ConfB, ConfC, ConfD, ConfE, ConfF

Other illustrative examples are provided in Appendix B.

3.5. Chapter Summary

This chapter provided a characterization of model families by drawing an analogy with software families. In particular, it discussed software families and their associated variability in the space and time dimensions. Then these concepts were used as a basis to establish an analogy between software families and model families. The chapter also discussed the special characteristics of model families and the challenges associated with them at both the model level and the metamodel level. The chapter highlighted the need for generic models to represent and capture model families. Finally, the chapter defined the Spatio-

Temporal Annotation Language (STAL), used to support the representation of family members and distinguish between them in the union model.

Chapter 4. Formalization of Model Families

This chapter provides a graph-based formalization of model families and union models. The rest of this chapter is organized as follows. The rationale behind formalization is discussed in Section 4.1. Sections 4.2 and 4.3 elaborate on the formalization of models and metamodels as typed graphs with attributes and constraints. Section 4.4 provides a formalization of model families and their union models. Propositional encoding of models is discussed in Section 4.5. Section 4.6 provides a proof-of-concept example on the union of propositionally encoded model, and finally, Section 4.7 summarizes the chapter. A simpler version of the formalization in this chapter has been published in [189].

4.1. Why Formalization

This chapter provides a formalization of the fundamental concepts related to the problem of variability modeling in model families. The reasons why we intended to formalize these concepts are as follows [190]:

- Formalization helps to reduce the ambiguity of abstract terms by making them more concrete in the form of formal definitions. This in turn helps researchers to illustrate and analyze problems and solutions efficiently, which leads to an overall better understanding of the problem.
- With proper formalization, modeling problems and their solutions can be described *independently* from modeling languages and tools.
- Formalization enables the development of robust algorithms and tools through solving problems that are exact by nature and that have provable properties.

This chapter illustrates the use of graphs and type graphs to formalize basic (meta)modeling concepts. The definitions of graphs and type graphs are then used as a basis for further formal definitions of model families and their union models. To model more detailed aspects of model families, I extend the definitions of basic type(d) graphs with attributes (i.e., attributed type graphs, or *E-graphs*) [191] and constraints [194][195].

4.2. Graph-based Formalization of (Meta) Models.

This section defines graphs (G) and type graphs (TG), and also defines more advanced extensions to type graphs such as: attributed type graphs (ATG), as *E-graphs* in particular, and attributed type graph with *constraints*. In addition, graph morphism (*f*) and other graph theory-related notations that are necessary for the formalization provided in this thesis are defined. Figure 29 illustrates the basic relationships between models and metamodels and their graph representation. The definitions that follow are based on [191][192][196]-[198].

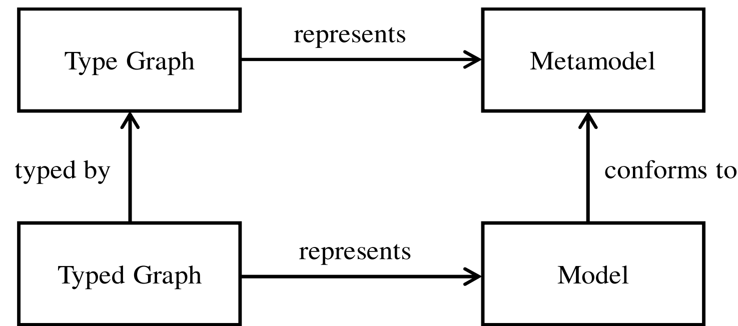


Figure 29 Relationship between models and their graph representation

4.2.1 Graphs and Graph Morphisms

This section defines the notions of graph and type graph, as well as how they are related by means of graph morphism.

Definition 1–Graph*:* A graph is a tuple $G = (N_G, E_G, src_G, tgt_G)$, where N_G is a set of graph nodes (or vertices), E_G is a set of graph edges, and functions $src_G, tgt_G: E_G \rightarrow N_G$ associate to each edge a source and a target node, respectively, such that $e : x \rightarrow y$ denotes an edge e with $src_G(e) = x$ and $tgt_G(e) = y$.

Definition 2– Graph morphism*:* let G and H be two graphs, a graph morphism $f: G \rightarrow H$ consists of a pair of functions (f_N, f_E) with $f_N: N_G \rightarrow N_H$ and $f_E: E_G \rightarrow E_H$, that preserves sources and targets of edges when composed ($\circ$), i.e. $f_N \circ src_G = src_H \circ f_E$ and $f_N \circ tgt_G = tgt_H \circ f_E$. In other words, for each edge $e_G \in E_G$, there is a corresponding edge $e_H = f_E(e_G) \in E_H$ such that $src_G(e_G)$ is mapped to $src_H(e_H)$ and $tgt_G(e_G)$ is mapped to $tgt_H(e_H)$. This is illustrated in Figure 30.

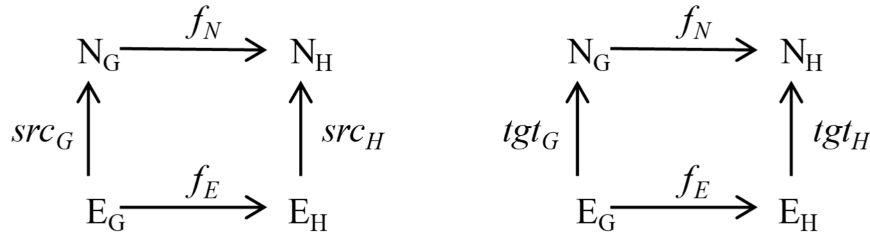


Figure 30 Graph morphism between graphs

4.2.2 Type(d) Graphs and Typed Graph Morphisms

In graph theory (as in typed programming languages, where each element is assigned a type), it is often useful to determine the well-formedness of a graph by checking whether it conforms to a so-called *type graph*. A type graph is a distinguished graph containing all the relevant types and their interrelations [199]. This is analogous to the relationship between models and metamodels in MDE, where each model (e.g., a UML design model) needs to conform to a metamodel (e.g., the UML metamodel). The correspondence between both concepts was already depicted in Figure 29.

Definition 3–Type graph (metamodel): a type graph TG is a distinguished graph, where $TG = (N_{TG}, E_{TG}, src_{TG}, tgt_{TG})$, and N_{TG} and E_{TG} are types of nodes and edges, respectively.

Definition 4–Typed graph (model): a typed graph is a triple $G_{typed} = (G, type, TG)$ such that G is a graph (Def. 1) and $type: G \rightarrow TG$ is a graph morphism (Def. 2) called the *typing morphism*. The typed graph G_{typed} is also called an instance graph of graph TG, and we write the set of all the instance graphs as $Inst[TG]$.

Example 1: Figure 31(b) shows an example of a *typed graph* (at the model level) typed over the *type graph* in Figure 31(a) (at one meta-level above). The type graph in this figure represents types of nodes and edges. There are two types of nodes here, namely State and Transition and two types of edges, namely src and tgt. In this thesis, node names and types in the typed graph are depicted inside the nodes in the form of name: type. For instance, in the node labelled with S1: State; S1 is the name of a node and State is the type of that node. Names and types of edges are represented in a similar manner.

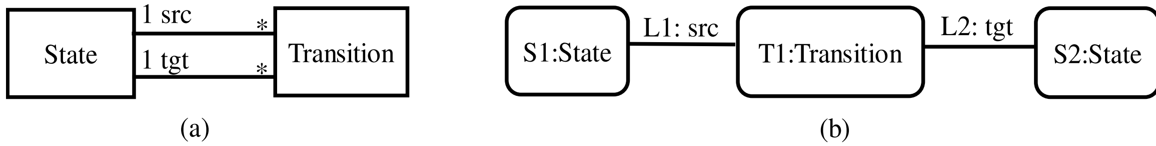


Figure 31 (a) Type graph, and (b) Typed graph

Definition 5–Typed graph morphism: Given a type graph *TG*, and two typed graphs G_{typed} and H_{typed} (typed over *TG*), a typed graph morphism $g: (G_{typed}, t_G: G_{typed} \rightarrow TG) \rightarrow (H_{typed}, t_H: H_{typed} \rightarrow TG)$ is a graph morphism $g: G_{typed} \rightarrow H_{typed}$ that also preserves typing, i.e., $g \circ t_H = t_G$, as illustrated in Figure 32.

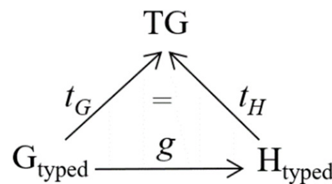


Figure 32 Typed graph morphism

4.2.3 Attributed Type(d) Graphs (as E-Graphs)

Given a typed graph (i.e., a model), it is useful in practice to attach additional information to nodes and edges by *attributing* them, such that each node and/or edge can contain zero or more attributes. In this case, we refer to *typed attributed graphs*, where attributes are typically a *name:value* pair that allows to attach a specific value to each attribute name. Values of attributes can be simple (e.g., a number or a string) or more complex (e.g., a Java expression) [198].

Example 2: As an example of a typed attributed graph, reconsider the typed graph in Figure 31(b) extended with attributes as in Figure 33, where Transition and State nodes have attributes respectively named tname and sname.

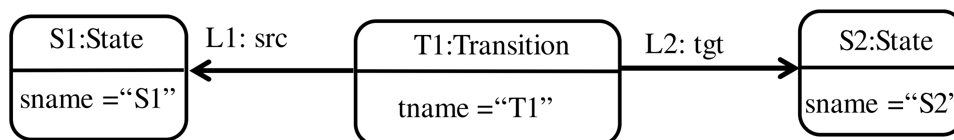


Figure 33 A typed attributed graph typed over the ATG in Figure 34

Given a typed attributed graph that is typed over some *TG* (i.e., a metamodel), that *TG* needs to constrain the names and types of attributes that are allowed for certain types of nodes and edges. In this context, we refer to an *Attributed Type Graph (ATG)*, as shown in Figure 34, which extends the type graph illustrated in Figure 31(a) with node attributes.

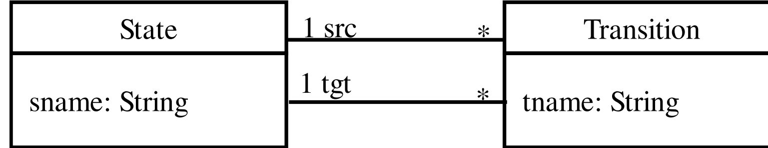


Figure 34 An attributed type graph (ATG)

Unlike graphs with node attributes only, edges in ATGs can also be provided with attributes, resulting in a new kind of graphs called *E-graphs*. In this thesis, we adopt the definition of Ehrig et al. [200] **for E-graphs to represent ATGs**[6], which allows attribution for both nodes and edges, where attributes of nodes (resp. edges) are represented as *special edges* between graph nodes (resp. graph edges) and data nodes that represent the data types of these attributes. Figure 35 visualizes the concept of E-graph.

***Definition 6–E-graph*:** An E-graph is a tuple $EG = (N_G, N_D, E_G, E_{NA}, E_{EA}, (src_j, tgt_j)$ $j \in \{G, NA, EA\})$ where:

- N_G and N_D are graph nodes and data nodes, respectively;
- E_G, E_{NA}, E_{EA} are graph edges, node's attribute edges, and edge's attribute edges, respectively;
- src_j, tgt_j, where $j \in \{G, NA, EA\})$ are source and target functions that assign to each of the three categories of edges (i.e., E_G, E_{NA}, E_{EA}) a source and a target, as follows:
  - $src_G: E_G \rightarrow N_G$, $tgt_G: E_G \rightarrow N_G$ for graph edges;
  - $src_{NA}: E_{NA} \rightarrow N_G$, $tgt_{NA}: E_{NA} \rightarrow N_D$ for node's attribute edges;
  - $src_{EA}: E_{EA} \rightarrow E_G$, $tgt_{EA}: E_{EA} \rightarrow N_D$ for edge's attribute edges.

[6] From now on in this thesis, the concept of ATG means implicitly an attributed typed graph that is represented as an E-graph.

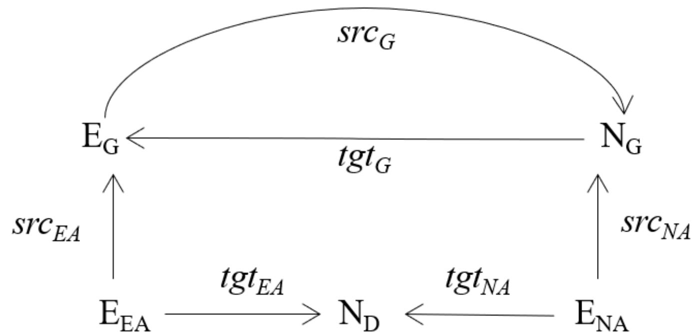


Figure 35 A visualization of an E-graph

Example 3: the ATG (in Figure 34) is represented as an E-graph (EG), as illustrated Figure 36. The figure shows the different categories of nodes and edges defined in Def. 6, namely:

- Graph nodes (N_G): State, Transition
- Data nodes (N_D): String
- Graph edges (E_G): src, tgt
- Node's attribute edges (E_{NA}): sname, tname

In this EG, attributes of nodes are represented as special edges between graph nodes (N_G) and data nodes (N_D), where N_D represents the type of an attribute. For example, the attribute tname is represented as an edge between the graph node Transition and the data node String. Attributes of edges are not shown in this example, but they can be represented in the same manner. In this example, we can consider the multiplicities of src, tgt association edges as attributes of these edges.

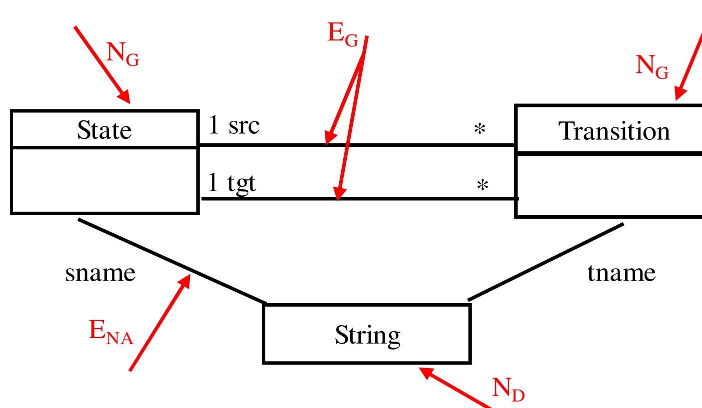


Figure 36 An E-graph (EG) with node attributes represented as edges

An instance typed graph (i.e., model) of the attributed E-graph in Figure 36 is illustrated in Figure 37, where this figure is a representation of the typed attributed graph found in Figure 33.

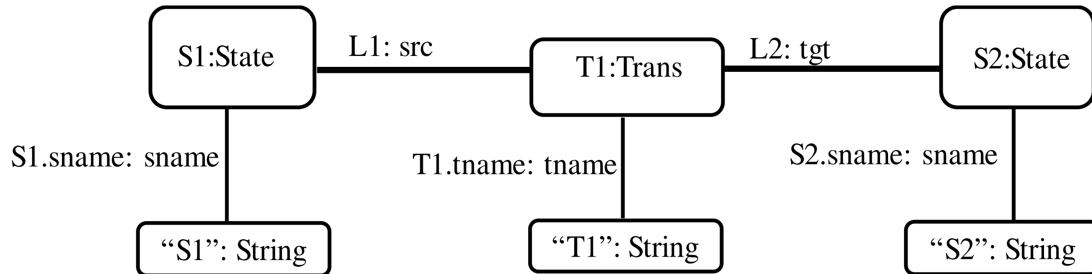


Figure 37 An instance typed attributed graph of the E-graph in Figure 36

4.2.4 Constraints

The concepts and definitions introduced so far are only related to graph *typing* (with *attribution*) without any further restrictions. In modeling languages, however, it is usually desired to restrict the set of valid instance models beyond typing. This can be achieved by using language constraints. Such constraints are either integrated internally into the language (such as multiplicity constraints) or attached explicitly (using OCL). In the literature, several approaches for model formalization with constraints were discussed, such as those found in [191][194][195]. One of the common ways to formalize constraints (that is also used in this thesis) is *graph constraints* [201].

In this section, I adapt the definitions of Kleppe and Rensink [201] to add some graph constraints to type graphs and, hence, to constrain the set of valid instance graphs. This thesis considers only multiplicity constraints for both attributes and association edges. Other graph constraints such as containment, abstractness, acyclicity [202], and external OCL constraints are left for future work.

First, I give a general definition of a constraint set over a graph, then I define the multiplicity constraints that I consider for the formalization of graphs.

***Definition 7–Graph Constraint*:** Let *ATG* be an E-graph. A constraint set over *ATG* is a tuple *(Cons, sat)*, where:

- *Cons* is a set of graph constraints;

- $sat \subseteq Inst\,[ATG] \times Con$ is a satisfaction relation over the instances of ATG.

We write G *sat* C to denote that an instance graph G satisfies a constraint $C \in Cons$.

Multiplicity Constraints

Multiplicity constraints regulate how many instances of an incident edge type (or attribute) are allowed per instance of a node type. To formalise multiplicity constraints, I first define what multiplicities are, and then introduce the corresponding graph constraint type.

Generally speaking, multiplicities are defined as ordered pairs of natural numbers in the form of *[lower, upper]*, where the second number (i.e., *upper*) is larger than or equal to *lower* or may take the special value * to indicate unboundedness. A number is within the range of the multiplicity if it is higher than or equal to the lower bound and smaller than or equal to the upper bound.

Definition 8–Multiplicity: A multiplicity is a pair $Mult = [l, u]$, where l and $u \in \mathbb{N} \cup \{*\}$ with $l \leq u$ or $u = *$. To retrieve the lower and upper bounds of multiplicities, we use the functions $low, upp: Mult \rightarrow \mathbb{N} \cup \{*\}$, such that if $x = [l, u]$ then $low(x) = l$ and $upp(x) = u$.

The multiplicity defined as above can be found at the *source end* (written in this thesis as *src*) or the *target end* (written as *tgt*) of an association edge to represent the potential degree of sharing of a node. In this thesis, we consider only multiplicities of ***target*** *association ends*[7] and refer to them simply as the *multiplicity constraint of an edge*, as formalized in Def. 9.

Definition 9–Multiplicity Constraint of a graph edge: Let *ATG* be an attributed type graph (represented as an E-graph). A multiplicity constraint of a graph edge e in *ATG* is a pair $edgeMult = (e, m)$ with $e \in E_G$ and $m \in Mult$, where $Mult = [l, u]$, l and $u \in \mathbb{N} \cup \{*\}$.

Satisfaction of multiplicity constraint of edges is denoted as $G \; sat \; edgeMult \Leftrightarrow \forall n{:}src(e),\; low(m) \leq |out\,(n,e)| \leq upp(m)$, where out$(n, e)$ is the multiplicity of the target end of association edge e.

Multiplicities can also be defined for *attributes* to regulate how many instances of a specific data type are allowed per instance of an attribute. Multiplicity of attributes can

[7] The rationale behind focusing only on the multiplicities of target association ends is discussed in Chapter 7. In a nutshell, there is only a need to relax multiplicity upper bounds in metamodels.

be viewed as a special type of the multiplicity defined in Def. 9, where $l=u \in \mathbb{N}$, and is defined as:

Definition 10–Multiplicity Constraint of attributes: Let *ATG* be an attributed type graph (represented as an E-graph). A multiplicity constraint of attributes of nodes (and edges) in *ATG* is a pair $attrMult=(attr, m)$ with $attr \in \{E_{NA}, E_{EA}\}$ and $m \in \mathbb{N}$.

Satisfaction of multiplicity constraint of attributes is denoted as G *sat* $attrMult \Leftrightarrow \forall n{:}src(attr), |out\,(n,attr)| = m$, where $out(n, attr)$ is the target multiplicity of $attr \in \{E_{NA}, E_{EA}\}$.

4.3. (Meta)models as Attributed Type(d) Graphs with Constraints

Based on Defs. 1-10, we can now formalize metamodels as attributed type graphs with constraints, and models as attributed typed graphs that satisfy these constraints.

Definition 11–Metamodel: A metamodel is a pair $MM=(ATG, Cons)$, where:

- *ATG* is an attributed type graph (represented as an E-graph).
- *Cons* is a constraint set in *ATG*.

Definition 12–Model: Given a metamodel MM, a model is a tuple $M = (G_{typed}, type, ATG, sat, Cons)$, where:

- $G_{typed} \in Inst[ATG]$, as in Def. 4.
- *type*: $G_{typed} \rightarrow ATG$, and;
- G_{typed} *sat Cons*

The set of instance models M of a metamodel MM is denoted as *Inst[MM]*, and we say that model M is an instance of MM if $M \in Inst[MM]$, and $\forall C \in Cons \mid M$ *sat* C.

4.4. Formalization of Model Families and Union Models

4.4.1 Formalization of Model Families

As previously mentioned in Section 3.2 and Figure 21, a model family consists of an arbitrary set of homogenous models that are instantiated from the same metamodel. In that

sense, we can define a model family as a set of typed attributed graphs that are instances of the same attributed type graph and satisfy the constraints of that type graph. In addition, the typed attributed graphs of a model family are related to each other by a *model family morphism relationship* illustrated in Figure 38. That is:

Definition 13–Model Family*: A model family* is a tuple *MF= ({G_{typed}}, type, ATG, sat, Cons)*, where *{G_{typed}}* ∈ Inst[ATG], type: {G_{typed}}→ATG, and {G_{typed}} *sat Cons.*

Definition 14–Model Family Morphism: Given an attributed type graph *ATG*, and a model family *MF={G, H, I, …, N}*, a model family morphism *g: (G, $type_G$: G→ATG) → (H, $type_H$: H→ATG) → (I, $type_I$: I→ATG) →…→ (N, $type_N$: N→ATG)* is a composition of typed graphs morphisms *g○h○i: G→N*

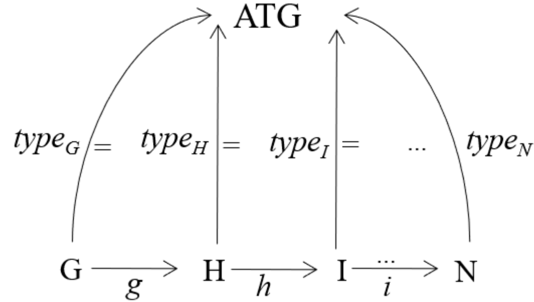


Figure 38 Model family morphism

Intuitively, since a model family is a set of typed graphs, then a the model family morphism represented above is basically a composition of typed graph morphisms, as defined previously in Def. 5 and illustrated in Figure 32. In other words, the model family morphism in Figure 38 is equivalent to the following typed graph morphisms (**Figure 39**), composed together.

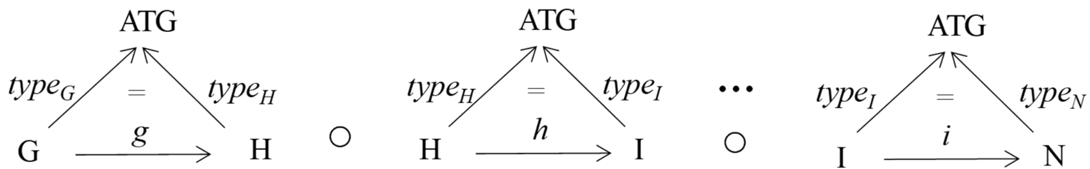


Figure 39 Model family morphism as a composition of typed graph morphisms

4.4.2 Formalization of Union Models

Semantically, a union model M_U of a given model family (MF) is the union of all elements *e*, in all individual models of that family. Def. 15 provides a formalized definition of M_U as a union of the graph representation of family members.

Definition 15–Union Model (M_U): let MF be a model family with two models, such that MF= *({G1, G2}, type, ATG, sat, Cons),* where type: $G_1, G_2 \rightarrow ATG$ (represented as an E-Graph), and G_1, G_2 *sat Cons*, and where $G1 = (N_{G1}, N_{D_G1}, E_{G1}, E_{NA_G1}, E_{EA_G1}, (src_j, tgt_j)\ j \in \{G, NA, EA\}, type_{G1})$ and $G2 = (N_{G2}, N_{D_G2}, E_{G2}, E_{NA_G2}, E_{EA_G2}, (src_j, tgt_j)\ j \in \{G, NA, EA\}, type_{G2})$ satisfy the following conditions:

- *Cond. 1*: If two nodes have the same name[8] and the same type, then these nodes are considered identical.
- *Cond. 2*: If two edges have the same name and the same type, and if they connect between the same source and target nodes, then these edges are considered identical.

Then the union model that represents *MF* is a model $M_U = G1 \cup G2 = (N_U, E_U, src_U, tgt_U, type_U)$, such that $N_U = (N_{G1} \cup N_{G2} \cup N_{D_G1} \cup N_{D_G2})$ and $E_U = (E_{G1} \cup E_{G2} \cup E_{NA_G1} \cup E_{NA_G2} \cup E_{EA_G1} \cup E_{EA_G2})$ and the functions src_U, tgt_U, and $type_U$ are:

$$src_U(e) = \begin{cases} src_{G1}(e), if\ e \in E_{G1} \\ src_{NA_G1}(e), if\ e \in E_{NA_G1} \\ src_{EA_G1}(e), if\ e \in E_{EA_G1} \\ src_{G2}(e), if\ e \in E_{G2} \\ src_{NA_G2}(e), if\ e \in E_{NA_G2} \\ src_{EA_G2}(e), if\ e \in E_{EA_G2} \end{cases}$$

[8] We assume that each node and each edge has its own unique identifier. For simplicity, we express this identity by means of a unique name.

$$tgt_U(e) = \begin{cases} tgt_{G1}(e), if\ e \in E_{G1} \\ tgt_{NA_G1}(e), if\ e \in E_{NA_G1} \\ tgt_{EA_G1}(e), if\ e \in E_{EA_G1} \\ tgt_{G2}(e), if\ e \in E_{G2} \\ tgt_{NA_G2}(e), if\ e \in E_{NA_G2} \\ tgt_{EA_G2}(e), if\ e \in E_{EA_G2} \end{cases}$$

$$type_U(elem) = \begin{cases} type_{G1}(elem), if\ elem \in N_{G1} \cup N_{D_G1} \cup E_{G1} \cup\ E_{NA_G1}\ \cup\ E_{EA_G1} \\ type_{G2}(elem), if\ elem \in N_{G2} \cup N_{D_G2} \cup E_{G2} \cup\ E_{NA_G2}\ \cup\ E_{EA_G2} \end{cases}$$

Provided that conditions Cond. 1 and Cond. 2 in Def. 15 are respected, the union operation can be generalized into an MF of any arbitrary size. That is, given MF= *({G1, G2, …, Gn}, type, ATG, sat, Cons),* its union model is $M_U = G1 \cup G2 \cup G3 \ldots \cup Gn$.

The union operation is incremental. That is, given a union model M_U already constructed for a particular family, then any upcoming model M_i added to that family is unified incrementally with M_U, such that the new union model becomes $M_{Unew} = M_U \cup M_i$. The annotations of M_U and M_i are also unified as discussed in Section 4.5. Consequently, the incremental nature of the union operation allows the merging of two or more individual models, a model and a union model, or two union models.

It is important to mention here that even if the typed graphs used to construct models are well-formed, there is no guarantee that their M_U will also be a well-formed model. In fact, M_U will respect the *typing constraints* imposed by the *ATG*, but maybe not the other constraints such as multiplicities of attributes and/or association ends (or other external OCL constraints). This issue is discussed in Chapter 6.

4.5. Propositional Encoding Language with Annotation (PELA)

In order to facilitate reasoning about models, and also to realize a simple graph union in practice, we encode typed graphs (i.e., models) as logical propositions. Such encoding hence provides a concrete syntax for defining models and metamodels. To encode a model, *m* into propositional logic, we need first to map elements in *m* into propositional variables and then join them. To achieve this, we propose a *propositional encoding language with annotations* (PELA), which defines specific naming conventions for the propositional

encoding of variables, where propositions themselves are annotated with STAL (see Section 3.4). While defining the syntax of PELA, we took into consideration that this language should be reversable. That is, a modeler should be able to retrieve (or to decode) a model back from its propositional encoding. This operation, however, is not supported by tools in this thesis, and such tool support, albeit not difficult to implement, is left for future work. The propositional encoding of models using PELA is defined as follows:

Definition 16–ElementToPropositionWithAnnotation: Given a model *M=(G, type, ATG, sat, Cons)*, where $G= (N_G, N_D, E_G, E_{NA}, (src_j, tgt_j), j\in\{G, NA, EA\})$[9], together with a STAL annotation specifying version numbers and configuration information, the mapping of elements of *M* into propositions with annotation *ElementToPropositionWithAnnotation(elem)* is defined according the following *syntactical* rules (along with their *semantics*):

- A *graph node* $n\in N_G$ of type $t\in N_{ATG}$ is mapped into a propositional variable *"n–t–<ver_{num}, $conf_{info}$>"* to express the semantics: "a model (with ver_{num} *and* $conf_{info}$) contains a node *n* of type *t*". Formally: $n–t$ *iff* $\exists\ n\in N_G \wedge type(n)=t$
- A *graph edge* $e\in E_G$ of type $t\in E_{ATG}$ with source node *x* and target node *y* is mapped into a propositional variable *"e–x–y–t–<ver_{num}, $conf_{info}$>"* to express the semantics: "a model (with ver_{num} *and* $conf_{info}$) contains an edge *e* of type *t* from node *x* to node *y*". Formally: $e–x–y–t$ *iff* $\exists\ e\in E_G \wedge type(e)=t \wedge src_G(e)=x \wedge tgt_G(e)=y$.
- A *data node* $dn\in N_D$ of type $t\in dataType$ owned by a graph node $n\in N_G$ is mapped into a propositional variable *"dn–n–t–<ver_{num}, $conf_{info}$>"* to express the semantics: "a model (with ver_{num} *and* $conf_{info}$) contains a node *n* that owns a data node *dn* of type *t*". Formally: $dn–n–t$ *iff* $\exists\ n\in N_G \wedge owner(dn)=n \wedge type(dn)=t$.
- A *node attribute edge* $nae\in E_{NA}$ of type $t\in attribute_name$ that is represented as a special edge between a graph node $n\in N_G$ and a data node $dn\in N_D$, where *n* is also the owner of that attribute, is mapped into a propositional variable "*nae–n–n–dn–t–<ver_{num}, $conf_{info}$>*" to express the semantics: "a model (with ver_{num} *and* $conf_{info}$) contains a node *n* which owns an attribute *nae* of type *t*, and this *nae* is represented as an *edge* from graph node *n* to data node *dn*". Formally: $nae–n–n–dn–t$ *iff* $\exists nae\in E_{NA} \wedge owner(nae)=n \wedge type(nae)=t \wedge src(nae)=n \wedge tgt(nae)=dn$. It is worth

[9] See Definition 6

clarifying that the "*n–n*" part in the pattern *nae–n–n–dn–t* represents the same element *n*. The first *n* indicates the owner of the attribute, and the second *n* indicates that this owner is also a source of the edge. We used this syntax to distinguish the node attribute edge (as a special edge) from the ordinary graph edge.

For example, if model M in Figure 37 is the second version of an initial model (say, M_0) and also represents configuration X, then the propositional encoding of state S1 in that model is: *S1–State–*$<ver_2, conf_X>$ and the propositional encoding of a node attribute edge S1.sname is *S1.sname–S1–S1–"S1"–name–*$<ver_2, conf_X>$.

It is important to emphasize here that the model M could also be a union model, rather than an individual model. According to Def. 15, since a union model M_U is a model, then the propositional encoding rules discussed above should also be applicable to M_U, except that the annotations of M_U elements are expressed as full STAL annotations, instead of single model annotations. This means that the four rules stated in Def. 16 preserve their syntax and semantics when applied to M_U, except that the annotation format changes from single ver_{nums} and single $conf_{info}$ into ranges, sets, or lists of ver_{num}, and lists of $conf_{info}$, expressed according to the grammar of STAL discussed in Appendix B.

Based on Def. 16, we can now define the mapping of an entire graph to propositions (with annotations) as follows:

***Definition 17–GraphToPropositionWithAnnotation**:* Given a typed graph *G*, a mapping of G's elements into a set of propositions is *GraphToPropositionWithAnnotation(G)=* *{ElementToPropositionWithAnnotation(elem)* | *elem* $\in N_G \cup N_D \cup E_G \cup E_{NA} \cup E_{EA}$ *}*

For example, the propositional encoding of the model in Figure 37 (repeated here for convenience) is shown to the left of Figure 40. We also assume that the model here represents the second version of a given initial model and represents configuration X. That is, each of its elements is annotated with $<ver_2, conf_X>$.

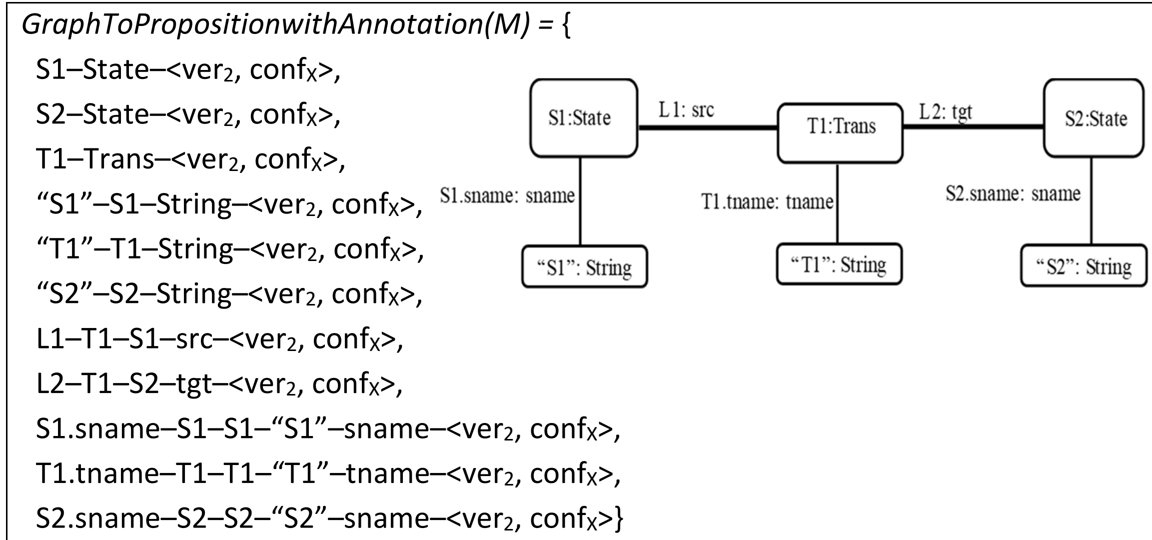


Figure 40 *GraphToPropositionwithAnnotation* encoding of model M in Figure 37

As mentioned previously, the process of propositional encoding of models with PELA is designed to be reversable. This means that given a propositional encoding with annotations of a model M, this model can be reconstructed back using the syntax conventions of PELA, as follows:

- For each propositional variable whose name fits the pattern n–t–$<ver_{num}, conf_{info}>$, we create a graph node n of type t, whose annotation is $<ver_{num}, conf_{info}>$.
- For each propositional variable whose name follows the pattern e–x–y–t–$<ver_{num}, conf_{info}>$, we create a graph edge e of type t between source node x and target node y, with annotation $<ver_{num}, conf_{info}>$.
- For each propositional variable whose name follows the pattern dn–n–t– $<ver_{num}, conf_{info}>$, we create a data node dn of type t, owned by a graph node n and annotated with $<ver_{num}, conf_{info}>$.
- For each propositional variable whose name follows the pattern nae–n–n–dn–t– $<ver_{num}, conf_{info}>$, we create a node attribute edge nae of type t, from graph node n (where n is also the owner of nae) to data node dn, and annotated with $<ver_{num}, conf_{info}>$.

4.6. Union of Propositional Encodings of Models

Given the propositional encoding of models discussed in Section 4.5 and Def. 17, the union operation simply becomes the union of the propositional encodings of individual models, as follows:

Definition 18–Proposition Encoding Union (PE_U): Let *MF* be a model family of two models *G1* and *G2* (Def. 14), where *G1* and *G2* are typed graphs with the same metamodel *ATG* and let *GraphToPropositionWithAnnotation(G1)* and *GraphToPropositionWithAnnotation(G2)* be their propositional encodings (Def. 17), then the union of the propositional encodings with annotations of G1 and G2 is: PE_U = *GraphToPropositionWithAnnotation(G1)* ∪ *GraphToPropositionWithAnnotation(G2).*

We can generalize the above definition to a set of arbitrary encoded models, where the union of the propositionally encoded models is annotated according to the grammar of STAL (Appendix B), and as discussed in Section 3.4.

As mentioned previously in Section 4.4.2, since the union operation is incremental, then a proposition encoding union (PE_U) can also be unified with other individual models, or even with other PE_{Us}. For instance, given a PE_U of a set of propositionally encoded models, and a new model M_i encoded as *GraphToPropositionWithAnnotation(M_i),* their union becomes $PE_{Unew} = PE_U \cup M_i$. The annotations of PE_U and M_i are also unified as discussed in Section 4.5, using the MergeSTAL algorithm discussed in Section 3.4.3.

4.6.1 Example

This section provides a simple, yet complete example for two versions of state transition diagrams, *M1* and *M2*. The example illustrates the formalization of both models as E-graphs. In addition, it illustrates their encoding into propositional variables as well as their union. In this example, *M1* and *M2* are assumed to be the first and the second version of a model, that also represent configuration A. Hence, *M1*'s and *M2*'s elements are respectively annotated with <ver_1, $conf_A$> and <ver_2, $conf_A$>.

Figure 41 represents M1 in the conventional representation of state transition diagrams. Figure 42 illustrates the representation of M1 as a canonical typed attributed E-graph and Figure 43 represents M1's propositional encoding.

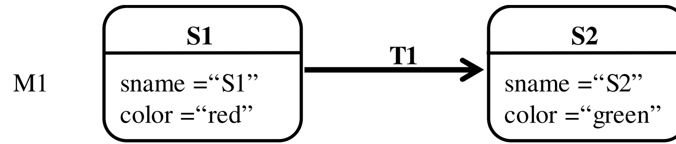


Figure 41 First version of a state transition diagram, M1

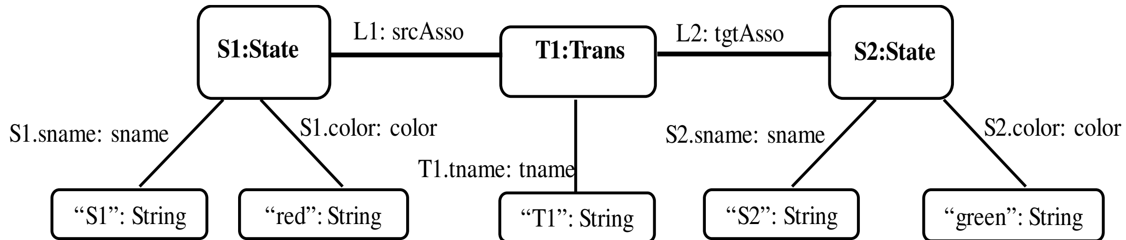


Figure 42 Representation of M1 as an E-graph

GraphToPropositionwithAnnotation(M1) = {
S1–State–$<ver_1, conf_A>$, S2–State–$<ver_1, conf_A>$,
“S1”–S1–String–$<ver_1, conf_A>$, “red”–S1–String–$<ver_1, conf_A>$,
“S2”–S2–String–$<ver_1, conf_A>$, “green”–S2–String–$<ver_1, conf_A>$,
T1–Trans–$<ver_1, conf_A>$, “T1”–T1–String–$<ver_1, conf_A>$,
L1–T1–S1–srcAsso–$<ver_1, conf_A>$, L2–T1–S2–tgtAsso–$<ver_1, conf_A>$,
S1.sname–S1–S1–“S1”–sname–$<ver_1, conf_A>$,
S1.color–S1–S1– “red”–color–$<ver_1, conf_A>$,
S2.sname–S2–S2– “S2”–sname–$<ver_1, conf_A>$,
S2.color–S2–S2–“green”–color–$<ver_1, conf_A>$,
T1.tname–T1–T1– “T1”–tname–$<ver_1, conf_A>$ }

Figure 43 Propositional encoding of M1

In the same manner, Figure 44 shows the conventional representation of *M2* as a state transition diagram, Figure 45 represents it as an E-graph, and Figure 46 demonstrates its propositional encoding.

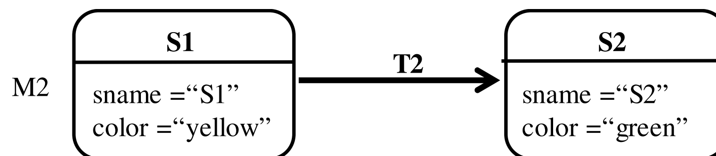


Figure 44 Second version of a state transition diagram, M2

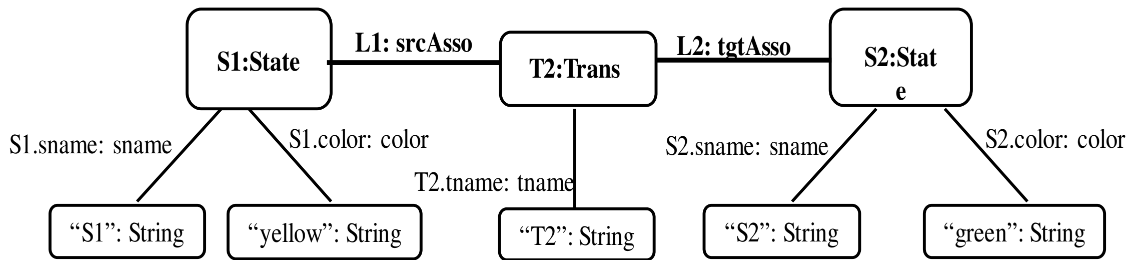


Figure 45 Representation of M2 as an E-graph

GraphToPropositionwithAnnotation(M2) = {
S1–State–$<ver_2, conf_A>$, S2–State–$<ver_2, conf_A>$,
"S1"–S1–String–$<ver_2, conf_A>$, "yellow"–S1–String–$<ver_2, conf_A>$,
"S2"–S2–String–$<ver_2, conf_A>$, "green"–S2–String–$<ver_2, conf_A>$,
T2–Trans–$<ver_2, conf_A>$, "T2"–T2–String–$<ver_2, conf_A>$,
L1–T2–S1–srcAsso–$<ver_2, conf_A>$, L2–T2–S2–tgtAsso–$<ver_2, conf_A>$,
S1.sname–S1–S1–"S1"–sname–$<ver_2, conf_A>$,
S1.color–S1–S1–"yellow"–color–$<ver_2, conf_A>$,
S2.sname–S2–S2–"S2"–sname–$<ver_2, conf_A>$,
S2.color– S2–S2–"green"–color–$<ver_2, conf_A>$,
T2.tname–T2–T2–"T2"–tname–$<ver_2, conf_A>$ }

Figure 46 Propositional encoding of M2

After encoding *M1* and *M2*, we become ready to construct their union model as the union of their propositional encoding with annotation (Def. 18), as follows:

PE$_U$ = *GraphToPropositionWithAnnotation(G1)* ∪
GraphToPropositionWithAnnotation(G2) = {
S1–State–<ALL>, S2–State–<ALL>,
“S1”–S1–String–<ALL>,
“red”–S1–String–<ver$_1$, conf$_A$>, “yellow”–S1–String–<ver$_2$, conf$_A$>,
“S2”–S2–String–<ALL>, “green”–S2–String–<ALL>,
T1–Trans–<ver$_1$, conf$_A$>, “T1”–T1–String–<ver$_1$, conf$_A$>,
T2–Trans–<ver$_2$, conf$_A$>, “T2”–T2–String–<ver$_2$, conf$_A$>,
L1–T1–S1–srcAsso–<ver$_1$, conf$_A$>, L2–T1–S2–tgtAsso–<ver$_1$, conf$_A$>,
L1–T2–S1–srcAsso–<ver$_2$, conf$_A$>, L2–T2–S2–tgtAsso–<ver$_2$, conf$_A$>,
S1.sname–S1–S1–“ S1”–sname–<ALL>, S1.color–S1–S1–“red”–color–<ver$_1$, conf$_A$>,
S1.color–S1–S1–“yellow”–color–<ver$_2$, conf$_A$>, S2.sname–S2–S2–“S2”–sname–<ALL>,
S2.color–S2–S2–“green”–color–<ALL>,
T1.tname–T1–T1–“T1”–tname–<ver$_1$, conf$_A$>,
T2.tname–T2–T2–“T2”–tname–<ver$_2$, conf$_A$> }

Figure 47 Union of the propositional encodings of M1 and M2

The canonical representation of the union model as an E-graph, with annotations, is depicted in Figure 48, and its conventional representation as an ordinary state transition diagram (also with annotations) is shown Figure 49.

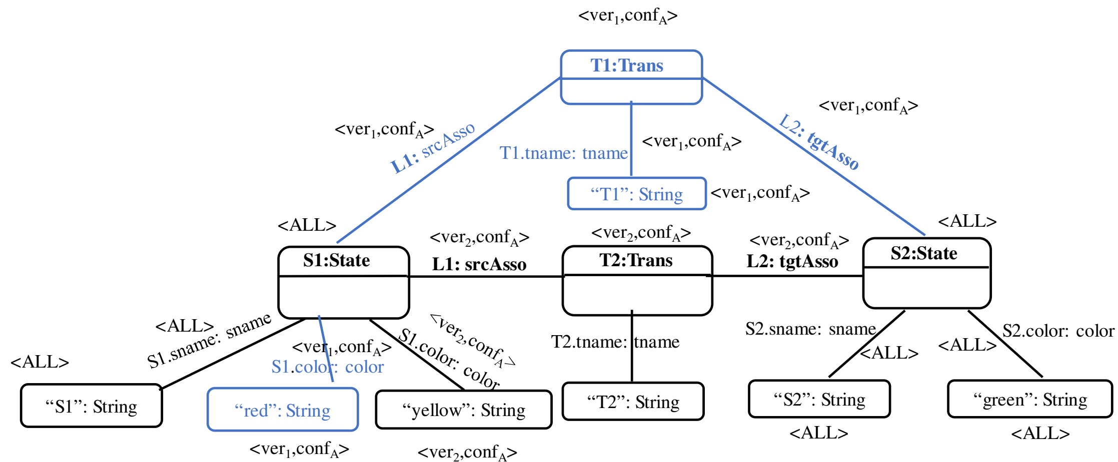


Figure 48 Representation of MU as an E-graph

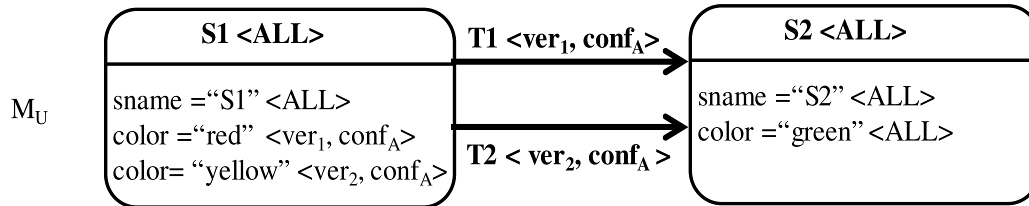


Figure 49 Conventional representation of MU as an annotated state transition diagram

4.6.2 Union of M_U and Another Model

After constructing a union model M_U, it is very possible that a new model, say M3 (or another union model of another subfamily) is added to the family. Assume that M3 (Figure 50 and Figure 51) represents the third version and configuration A of a model. Hence, M3's elements will be annotated as <ver_3, $conf_A$>. The propositional encoding, with annotation, of M3 is illustrated in Figure 52.

If we want to join M3 to the union model M_U constructed previously, this will be done incrementally. That is, instead of constructing the union of M1, M2 and M3 all over again, the new union model M_{Unew} would be equal to $M_U \cup M3$, expressed in terms of propositional encoding in Figure 53, and in the canonical form of state transition diagram in Figure 54.

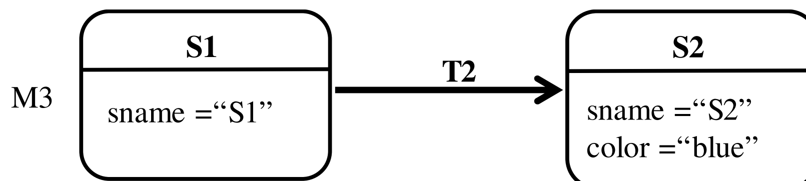


Figure 50 Third version of a state transition diagram, M3

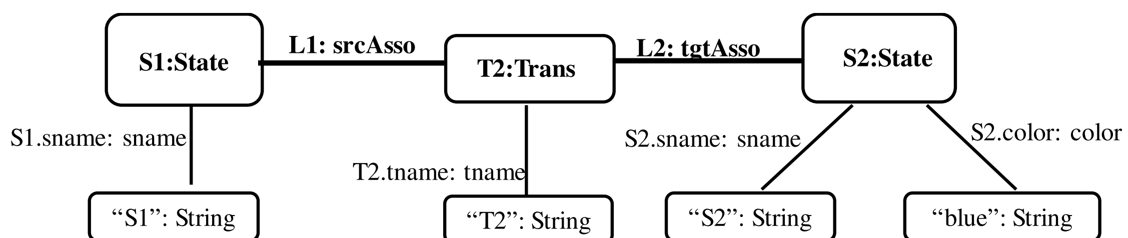


Figure 51 Representation of M3 as an E-graph

GraphToPropositionwithAnnotation(M3) = {
S1–State–<ver$_3$, conf$_A$>, S2–State–<ver$_3$, conf$_A$>,
“S1”–S1–String–<ver$_3$, conf$_A$>,
“S2”–S2–String–<ver$_3$, conf$_A$>, “blue”–S2–String–<ver$_3$, conf$_A$>,
T2–Trans–<ver$_3$, conf$_A$>, “S1”–S1–String–<ver$_3$, conf$_A$>,
“T2”–T2–String–<ver$_3$, conf$_A$>,
L1–T2–S1–srcAsso–<ver$_3$, conf$_A$>, L2–T2–S2–tgtAsso–<ver$_3$, conf$_A$>,
S1.sname–S1–S1–“S1”–sname–<ver$_3$, conf$_A$>,
S2.sname–S2–S2–“S2”–sname–<ver$_3$, conf$_A$>,
S2.color– S2–S2–“blue”–color–<ver$_3$, conf$_A$>,
T2.tname–T2–T2–“T2”–tname–<ver$_3$, conf$_A$> }

Figure 52 Propositional encoding of M3

PE$_{Unew}$ = PE$_U$ ∪ *GraphToPropositionWithAnnotation(G3)* = {
S1–State–<ALL>, S2–State–<ALL>,
“S1”–S1–String–<ALL>,
“red”–S1–String–<ver$_1$, conf$_A$>, “yellow”–S1–String–<ver$_2$, conf$_A$>,
“S2”–S2–String–<ALL>,
“green”–S2–String–<(ver$_1$, ver$_2$), conf$_A$>, “blue”–S2–String–<ver$_3$, conf$_A$>,
T1–Trans–<ver$_1$, conf$_A$>, T2–Trans–<(ver$_2$, ver$_3$), conf$_A$>,
“T1”–T1–String–<ver$_1$, conf$_A$>, “T2”–T2–String–<(ver$_2$, ver$_3$), conf$_A$>,
L1–T1–S1–srcAsso–<ver$_1$, conf$_A$>, L2–T1–S2–tgtAsso–<ver$_1$, conf$_A$>,
L1–T2–S1–srcAsso–<(ver$_2$, ver$_3$), conf$_A$>, L2–T2–S2–tgtAsso–<(ver$_2$, ver$_3$), conf$_A$ >,
S1.sname–S1–S1–“ S1”–sname–<ALL>,
S1.color–S1–S1–“red”–color–<ver$_1$, conf$_A$>,
S1.color–S1–S1–“yellow”–color–<ver$_2$, conf$_A$>,
S2.sname–S2–S2–“S2”–sname–<ALL>,
S2.color–S2–S2–“green”–color–<(ver$_1$, ver$_2$), conf$_A$>,
S2.color–S2–S2–“blue”–color–<ver$_3$, conf$_A$>,
T1.tname–T1–T1–“T1”–tname–<ver$_1$, conf$_A$>,
T2.tname–T2–T2–“T2”–tname–<(ver$_2$, ver$_3$), conf$_A$> }

Figure 53 Union of the propositional encodings of PEU and M3

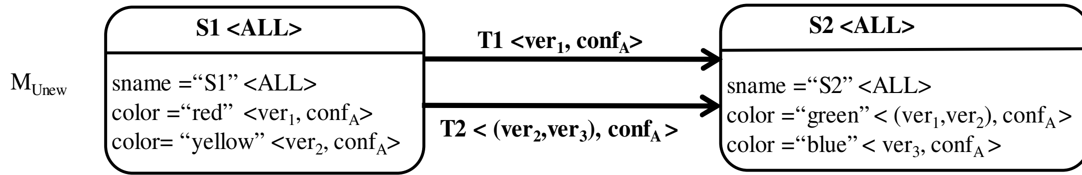


Figure 54 Conventional representation of MUnew as an annotated state transition diagram

4.6.3 Decisions while Constructing M_U:

It is important to emphasize here that while constructing the union model, those elements that have the same names and types, but different attributes, are still considered to be the same element with two different attribute sets.

In fact, given that the same elements could vary in their attributes during evolution or over variations, the union model that captures the family could be represented in different ways, depending on what we decide regarding these elements (i.e., whether to consider them as the same elements or as different elements). To narrow the number of possible representations of a union model, we took the following decisions while constructing a union model:

1. We consider the names of elements as unique identifiers (UID) for simplicity. However, any other UID could also be used.
2. If two elements have the same name and type, then they are considered as one element in M_U.
3. If two elements have the same name but different types, then they are considered as different element in M_U.
4. Elements with same names and types but different attributes are still considered as one element with multiple attributes in M_U.
5. If two edges have the same name and type, and connect between the same source and target nodes, then these edges are considered identical in the M_U.
6. If the source and target of the same edges are different, then edges are considered different in the M_U.
7. If the same pair of nodes is connected by multiple edges, then these edges should be different.

4.7. Chapter Summary

This chapter presented a formal specification of (meta)models, model families, and union models based on graph theory. We consider such a formal specification essential because it precisely describes the problem to be solved on a high level of abstraction. With the selected formalization, the creation of a union model becomes the simple union of individual models (or other model families, enabling incremental construction). In addition, based on the provided graph-based formalization, a propositional encoding language is provided as a syntax to describe models and their union models in a simple way, and to facilitate analysis and reasoning about models thereafter.

Chapter 5. Analysis and Reasoning with Model Families

This chapter is dedicated for the exploration of **RQ2**, which is: *How efficient is reasoning and analysis with a group of models, all at once, using M_U in comparison to the use of individual models?* The rest of this chapter is organized as follows: Section 5.1 defines three reasoning tasks to evaluate their performance using first, union models, and then using individual model several times. Section 5.2 discusses our experimental setup, methodology and implementation. Our empirical results are reported in Sections 5.3 to 5.5 . Section 5.6 highlights the potential threats to validity. Finally, Section 5.7 summarizes the chapter. A preliminary version this chapter has been published in [189].

5.1. Reasoning Tasks

To answer RQ2, we need to describe how a union model can facilitate analysis and reasoning with sets of models instead of only single models. To achieve this, we consider three *reasoning tasks* (RTs), namely *property checking* (which is already known in the literature), *trend analysis*, and *commonality analysis* (which we propose in this thesis). Then we compare the performance of the three RTs using M_U as opposed to using individual models.

Although these kinds of analyses can be performed using individual models (several times, one model at a time), our objective is to make these analyses more efficient using M_U. In addition, we aim to reduce the effort needed for loading each model into a tool, analyzing the model, saving the analysis results, and then moving to the next model, especially as this effort *cannot* be neglected with a large number of models. These manual steps are however not considered in our results, so our results and performance improvements are conservative.

5.1.1 RT1: Property Checking

Property checking on models aims to verify whether a model satisfies a particular property or not. Given a model *m* and a property *p*, the result of property checking is either True if

m satisfies *p*, or False otherwise. For instance, a modeler may want to check whether a group of state machine diagrams contains self-looping edges or not, or she may check whether there exist two or more different actors in a GRL model family that contain the same goal. In these scenarios, property checking is beneficial to help modelers understand, for example, what is common between model versions or variations that violate a property.

In this chapter, we limit ourselves to language-independent, syntactic properties (which describe the structure of models) other than semantic properties (which describe the behavior of models, e.g., traces). The rationale behind this scoping is because our approach aims to be applicable to any metamodel-based modeling language. However, while there exists a standard approach for defining the syntax of a modeling language (i.e., through metamodeling), there is no common approach for specifying semantics. So, we limit our approach to checking those properties related to a language syntax, independently from any language specificity. Hence, "property" here means "syntactic property". The next chapter will discuss and evaluate semantic aspects for one particular language.

To perform property checking, we assume that a property *p* (expressed in any constraint language such as FOL or OCL) can be grounded over the vocabulary of models. Hence, a corresponding propositional formula Φp can be obtained. For example, given a well-formedness constraint $\Phi c{:}\forall t{:}\ \texttt{Transition}\ \exists s{:}\texttt{State}|\texttt{t.src=s}$, it can be grounded over the vocabulary of the model in Figure 55 as follows:

$$\texttt{T1-S1-S2-Transition} \Rightarrow \texttt{S1-State} \wedge \texttt{T2-S2-S3-Transition} \Rightarrow \texttt{S2-State}$$

Figure 55 An example of propositional encoding of a property

As can be noted, the example considers the graphical representation of the state machine presented in the canonical form in Figure 31.b, where Transitions T1 and T2 are represented here as directed edges between states, and not as nodes.

Formally speaking, given the propositional encoding of both models (Section 4.5) and properties, the task of property checking can be defined as follows:

Definition 19–Property Checking: Given model *m* and a property *p*, and their propositional encodings Φm and Φp, respectively, we check if the expression $\Phi m \wedge \Phi p$ is satisfiable or not using a SAT solver.

For RT1, we checked the "*cyclic composition property*" inspired from [203], which ensures that "the model does not contain self-looping edges". It is important to highlight here that checking the "self-looping edges" property in individual models and in their M_U is a simple reasoning task that will always produce the same result. That is, if a self-looping edge exists in any of the individual models, then it will be detected in the M_U that captures these individual models, and vice versa. This case, however, is not necessarily true for other properties, such as, for example, the acyclicity property.

Figure 56 shows two simple state machine models that are acyclic (i.e., cycle-free), and their union model M_U that is not acyclic (i.e., has a cycle). In this scenario, checking the acyclicity property in individual models M1 and M2 will produce a different result than checking the same property in M_U. To address this issue, there is a need to explicitly consider annotations on elements during the property checking to determine the actual occurrence of cycles. For instance, if the link in model M1 is annotated with $<v_1>$, and the link in M2 is annotated with $<v_2>$, then both links in M_U will be different and will not be considered as an actual cycle (even though they seem to constitute a structural cycle). A real cycle only exists when the intersection of the annotations of the links involved is not empty.

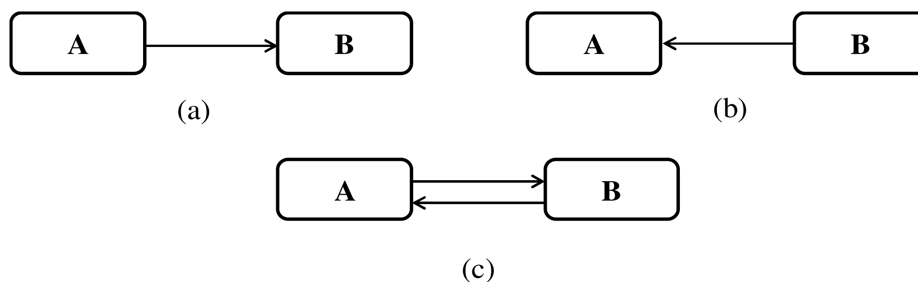


Figure 56 (a) An acyclic model M1, (b) an acyclic model M2, (c) a union model of M1 and M2 that is not acyclic

Property checking while considering annotations is more complicated and we need to further investigate its complexity. That is, we need to characterize the conditions under which

this kind of property checking is beneficial using M_U in comparison to using individual models, multiple times.

5.1.2 RT2: Trend Analysis

This analysis aims to search for a particular element across members of a model family and study the *trend* of that element, i.e., the behavior of elements over space/time. In other words, a trend analysis studies how properties of elements change over the course of time or across configurations. For instance, a modeler may need to search for a particular goal, say GoalX in all members of a GRL family to conduct a trend analysis about the properties of that goal (e.g., its importance value, or satisfaction value), and to observe how that value changes across model version/variations to get some insights about its evolution pattern.

5.1.3 RT3: Commonality Analysis

We suggest this type of analysis to enable modelers to check for those elements that are common in all (or part) of versions or variations of models in a family. This type of analysis is aligned with the commonality-based analysis in the SPL domain, where commonality is a key metric that indicates the reuse ratio of a feature across the SPL [204]. Following the same rationale, elements that are found in this analysis to be common among the majority of models can be inferred to be important. For example, if a modeler is investigating several design options of a particular system, and she needs to know which elements are important in design options, then she would conduct this analysis *once* using the M_U of the model family she has at hand (instead of doing a pairwise search on each version/variation of individual models).

5.2. Experiments

We assess the feasibility of reasoning using M_U empirically. We ran experiments with parameterized random inputs that simulate different settings of various reasoning and analysis categories. In this chapter, we build on the formalization of union models (Section 4.4.2) and use formalized GRL models and state machine models. Our approach is however not bound to these languages and is applicable to other metamodel-based languages.

5.2.1 Methodology

To evaluate the feasibility of using M_U with the three reasoning tasks RT1, RT2, and RT3, we first measured the total time (in seconds) needed to perform each one of the RTs on each individual model (one model at a time), and we refer to this time as T_{ind}. Then, we measured the time needed to accomplish the same task with M_U, and refer to this time as T_{MU}. Then, we compute performance improvements with metrics time *speedup* (as used in [205]), defined originally as $speedup = T_{ind} / T_{MU}$ and *time saving* (in minutes for RT1 and in seconds for RT2 and RT3) calculated as $TimeSaving = (T_{ind} - T_{MU})$.

We also define $T_{construct}$ as the time needed to construct M_U. Although $T_{construct}$ is usually quite small and can be performed once before being amortized over multiple analyses, I distinguish two categories of experiments, named **Exp.1** and **Exp.2**. In Exp.1, I consider $T_{construct}$, such that the speedup is calculated as $speedup_with_constrTime = T_{ind} / (T_{construct} + T_{MU})$. In Exp.2, on the other hand, I neglect the time needed to construct M_U, where the speedup is calculated here (same as in [205]) as $speedup_without_constrTime = T_{ind} / T_{MU}$. In both experiments, a speedup larger than 1 is a positive result, and the larger the speedup, the better the improvement.

The reason why $T_{construct}$ is considered here is to be fairer and more realistic in the experiments, especially for large models, where it becomes necessary not to neglect the time needed to construct M_U. Another reason is to compare the results of both categories of experiments to be able to reach a conclusion on whether to always neglect $T_{construct}$ or not. As mentioned, the time that an analyst would need to analyze models individually in a realistic context (by loading the model in a tool, performing the analysis, and saving the results) is not taken into consideration in T_{ind}.

For both experiments, I considered the following experimental parameters: (1) the size of individual models (*SIZE*), which represents the number of elements (i.e., nodes and edges) in each individual model and (2) the number of individual models in a model family (*INDV*). To control the possible combinations of parameters *SIZE* and *INDV* and to facilitate reporting, I followed the methodology proposed by Famelis et al. [175][205] to discretize the parameters' domain into categories, where the ranges of values for the *INDV* parameter were set based on pilot experiments conducted by Famelis et al. [175] to determine reasonable ranges of individual models that constitute a family. Regarding the ranges of

values for the *SIZE* parameter, we relied on our own experience with goal models, where models with a few dozen elements are considered to be small, while models with many hundreds or more elements are deemed to be extra large.

For parameter *SIZE* four categories were defined based on the number of nodes and edges, as follows: small (S), medium (M), large (L), and extra-large (XL). To calculate the ranges of each size category, we performed experiments with a seed sequence (0, 5, 10, 20, 40). The boundaries of each category were calculated from successive numbers of the seed sequence using the formula $n \times (n+1)$. Using the same formula, a representative exemplar of each category is calculated by setting *n* to be the median of two successive numbers in the seed sequence. We followed the same methodology for the number of individual models, *INDV*, using a seed sequence (0, 4, 8, 12, 16). The four size categories (S, M, L, XL) are shown in Table 3. The ranges of all categories of *SIZE* and *INDV* and the selected exemplars for each category are shown in Table 2. These ranges (generated from the seeds mentioned above) are in line with our own real experience dealing with goal models and state machines of various sizes. The same can be said for the number of individual models; a family is considered small when it contains a handful of models, but it is considered very large when it contains hundred of models or more.

Table 2 Categories of parameter *SIZE* (number of elements in a model)

#elements/model (*SIZE*)	(0, 30]	(30, 110]	(110, 420]	(420, 1640]
Exemplar	12	56	240	930
Category	S	M	L	XL

Table 3 Categories of parameter *INDV* (number of individual members in a family)

#of individual models (*INDV*)	(0, 20]	(20, 72]	(72, 156]	(156, 272]
Exemplar	6	42	110	210
Category	S	M	L	XL

To evaluate the property checking task (i.e., RT1), each annotated individual model *m* in a model family *MF* was encoded as a propositional logic formula, namely $\Phi m = \bigwedge ElementToPropositionWithAnnotation(ei),\ ei \in m$, where e_i are elements of

the model m. A union model M_U of that MF is also encoded as $\Phi M_U = \bigwedge ElementToPropositionWithAnnotation(ei),\ ei \in MU$. Furthermore, the property to be checked was encoded into a propositional formula Φp. Then, a SAT solver was used to check if the encodings of each of the individual model and their union model satisfy (or not) the property. In particular, for each individual model, a formula $\Phi m \wedge \Phi p$ is constructed. The property is said to hold in any model if and only if this formula is satisfiable. Similarly, A formula $\Phi M_U \wedge \Phi p$ is constructed and checked against whether the property is satisfiable. In both experiments (using the same computer settings), the time it took to check a property on individual models (T_{ind}) is recorded and compared to the time needed to do the check on union models (T_{MU}).

5.2.2 Implementation

To validate our approach, we used the NetworkX 2.2 Python library [206] to implement attributed typed graphs (according to Def. 5), and we implemented our own union algorithm on top of that library to construct M_U (based on Defs. 16 and 18). NetworkX 2.2 is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex graphs [206]. It is enriched with a variety of features from the support of graph data structures and algorithms to analysis measures to visualization options.

To have a family of state machine models, I used NetworkX's graph generators to randomly generate valid attributed typed graphs (with different parameters *SIZE* and *INDV*). These graphs correspond to typed state machines with *likely evolutions*, i.e., a sequence of random but typical manipulations on state machine models that leads to different versions. A sample of the generated graphs were manually checked to make sure that we are generating likely changes to existing models rather than generating completely independent models. Due to the complete random nature of the graph generators, we found that although the amount of changes on state machine models can be controlled, the topology of the resulting graph cannot. That is, the same node in a model could change its incoming and/or outgoing edges randomly, leading to a different model. Such deviations between individual models could lead to a union model with large variations, indicated by the number of annotations on each element. Although this is not the best, or even the typical case of state machine model families, we decide to perform experiments on those generated

models so as to examine the complex families, where the performance cannot likely get worse.

For GRL models, the generation of models was less random, where I took a set of real GRL models (illustrated in Chapter 6) as a starting point and enlarged them according to the different combinations of parameters *SIZE* and *INDV*. While growing the models, I created a set of random, but realistic modifications that involve adding and/or deleting intentional elements and/or element links or modifying their attributes. I then constructed G_U from the generated graphs using my union algorithm. G_U is the union of a set of typed graphs, and hence G_U corresponds to M_U.

For RT1, we checked the "*cyclic composition property*" inspired from [203], which ensures that "the model does not contain self-looping edges". A propositional formula (Φp:$\forall$e:E_G, src_G(e)!=tgt_G(e)) was also generated for this property. The propositional encodings were generated according to the rules discussed in Section 4.5, and they were fed as literals to the *MiniSAT* solver included in the *SATisPY* package [207]. SATisPy is a Python library that provides an interface to various SAT solver applications.

To build confidence about the property checking results, the graphs and their union model were tested to check for the existence of any self-looping edges. Output solutions retrieved for individual models of a particular family were compared to the solutions returned for the corresponding union model. The results were the same.

All experiments were executed on a laptop with an Intel Core i5-8250U (8th Gen) 1.6GHz quad-core CPU and 8GB RAM, running Windows 10-x64.

The next three sections are organized according to the experiments conducted to evaluate RT1, RT2, and RT3. All figures illustrated next represent a summary of the average results of 15 runs, represented for all *SIZE* and *INDV* categories together.

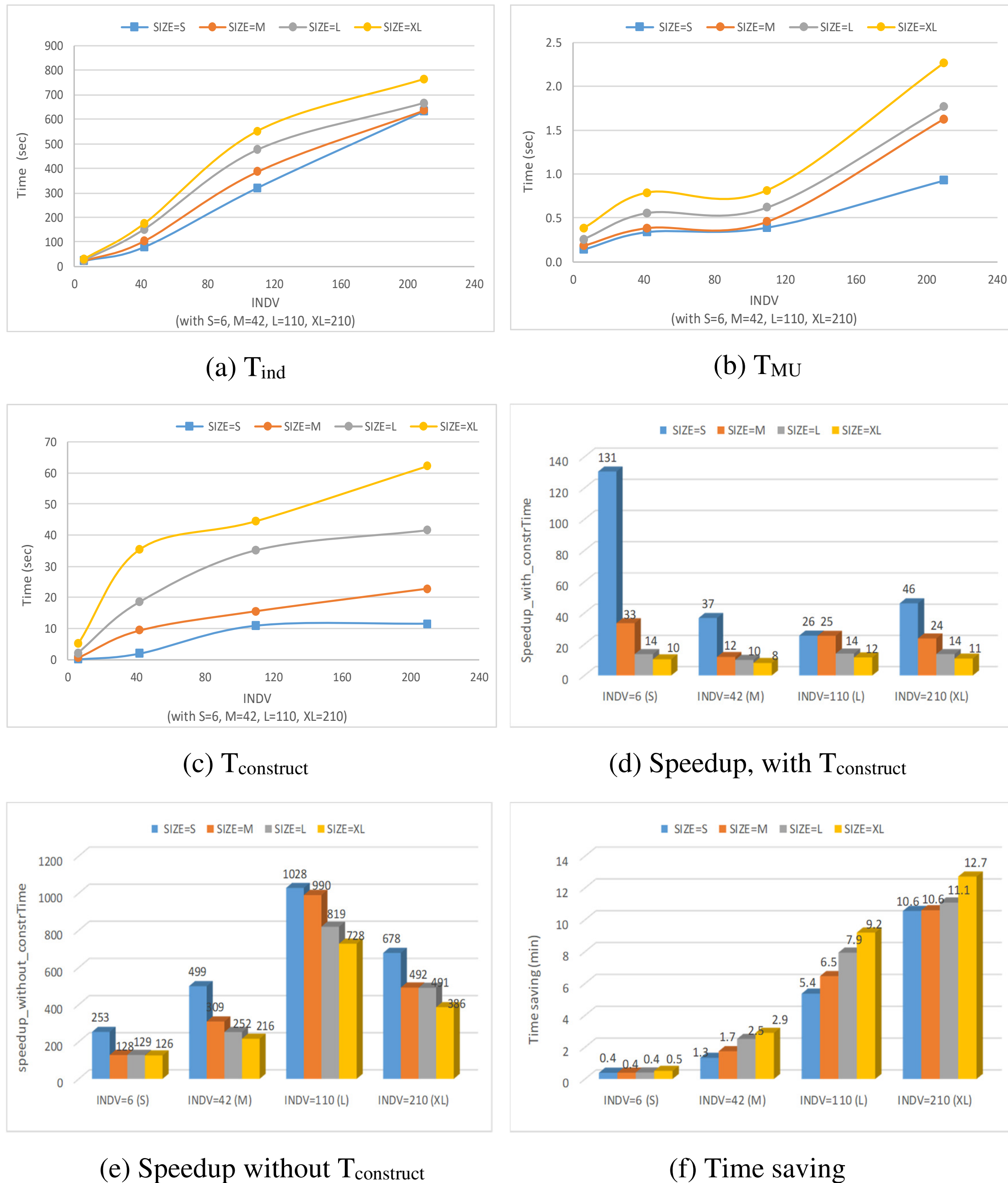
5.3. Results for Property Checking (RT1)

This section reports on our empirical results for the property checking reasoning task. Figure 57(a) illustrates T_{ind}, which is the total time of performing property checking on each individual model, for all *SIZE* and *INDV* categories. Figure 57(b) reports on T_{MU}, which is the time needed to perform property checking on union models that capture individual models of different *SIZE* and *INDV*. In addition, Figure 57(c) shows the time $T_{construct}$

needed to construct these union models. Figure 57(d) shows the time speedup with $T_{construct}$, i.e., *speedup_with_constrTime*= T_{ind} / ($T_{construct}$ + T_{MU}), while Figure 57(e) shows the speedup without considering $T_{construct}$, calculated as *speedup_without_constrTime* = T_{ind} / T_{MU}. Finally, Figure 57(f) highlights the time saving in minutes achieved by using M_U to perform property checking calculated as *TimeSaving*=(T_{ind}–T_{MU}).

Figure 57(a)-(c) shows that T_{ind}, T_{MU}, and $T_{construct}$ increase when the size of models (i.e., *SIZE*) or the number of models in a family (i.e., *INDV*) increase. In addition, it can be noticed that for each *SIZE* category, the increase of T_{ind} is important as the *INDV* parameter grows from *INDV*=S to *INDV*=XL. For instance, with *SIZE*=XL, we observe 30.85 second on average for *INDV*=S to 765.04 seconds on average for *INDV*=XL. On the other hand, for each *SIZE* category, the increase of T_{MU} as *INDV* grows is marginal. For example, for XL-sized models, T_{MU} increases from 0.39 seconds (for *INDV*=S) to 2.25 seconds (for *INDV*=XL).

Furthermore, it can be inferred from Figure 57(d)-(e) that the use of M_U for property checking achieves a noticeable time speedup compared to performing the same task on a set of individual models separately. For *speedup_with_constrTime* (Figure 57(d)), the highest speedup (=131) was observed with a small number of individual models (i.e., *INDV*=S) that are of a small size (i.e., *SIZE*=S). The smallest speedup (=8), was observed when *INDV*=M, and *SIZE*=XL. In addition, Figure 57(d) shows that for each *INDV* category, there is a noticeable pattern of speedup degradation as the number of elements per individual model (i.e., *SIZE*) increases. This is due in part to the increase of $T_{construct}$ as the *SIZE* increases. Nevertheless, the speedup never goes below 1, which means that even with very large models (with *INDV* =XL and *SIZE*=XL), the time to perform property checking on a group of such models (using M_U), considering the time to construct M_U, is still better than performing property checking on all individual models. Figure 57(e) shows that the time speedup becomes more significant when $T_{construct}$ is ignored, where the highest *speedup_without_constrTim*=1028 with models of *SIZE*=S and *INDV*=L. This high value of 1028, which is higher than the number of models in the family (110), is caused in part because of a low denominator value with low resolution (e.g., 1028 = 308.4/**0.3**), but also by the time taken by the (Python) environment to load before executing the code of each individual model, which is not neglectable particularly for small models.



(a) T_{ind} (b) T_{MU}

(c) $T_{construct}$ (d) Speedup, with $T_{construct}$

(e) Speedup without $T_{construct}$ (f) Time saving

Figure 57 Summary of results for property checking (RT1), for all *INDV* and *SIZE* categories

It is important to emphasize here that the erratic behaviour of the time speedup (with and without $T_{construct}$) across all categories of *SIZE* and *INDV* does not necessarily mean that one category is superior over the other. This is because speedup reflects the ratio between

T_{ind} and T_{MU} (and $T_{construct}$ in case of calculating *speedup_with_constrTime*), where it could happen that this ratio fluctuates in a random way.

It is not necessary that the speedup with *INDV*=S should be always better than that of *INDV*=M, *INDV*=L, etc., or vice versa, as the topology of the models generated may also influence efficiency (and this aspect is not controlled in this experiment). To this end, the speedup metric is used in this thesis to demonstrate improvements achieved by using M_U in general, regardless of the behavior of this improvement.

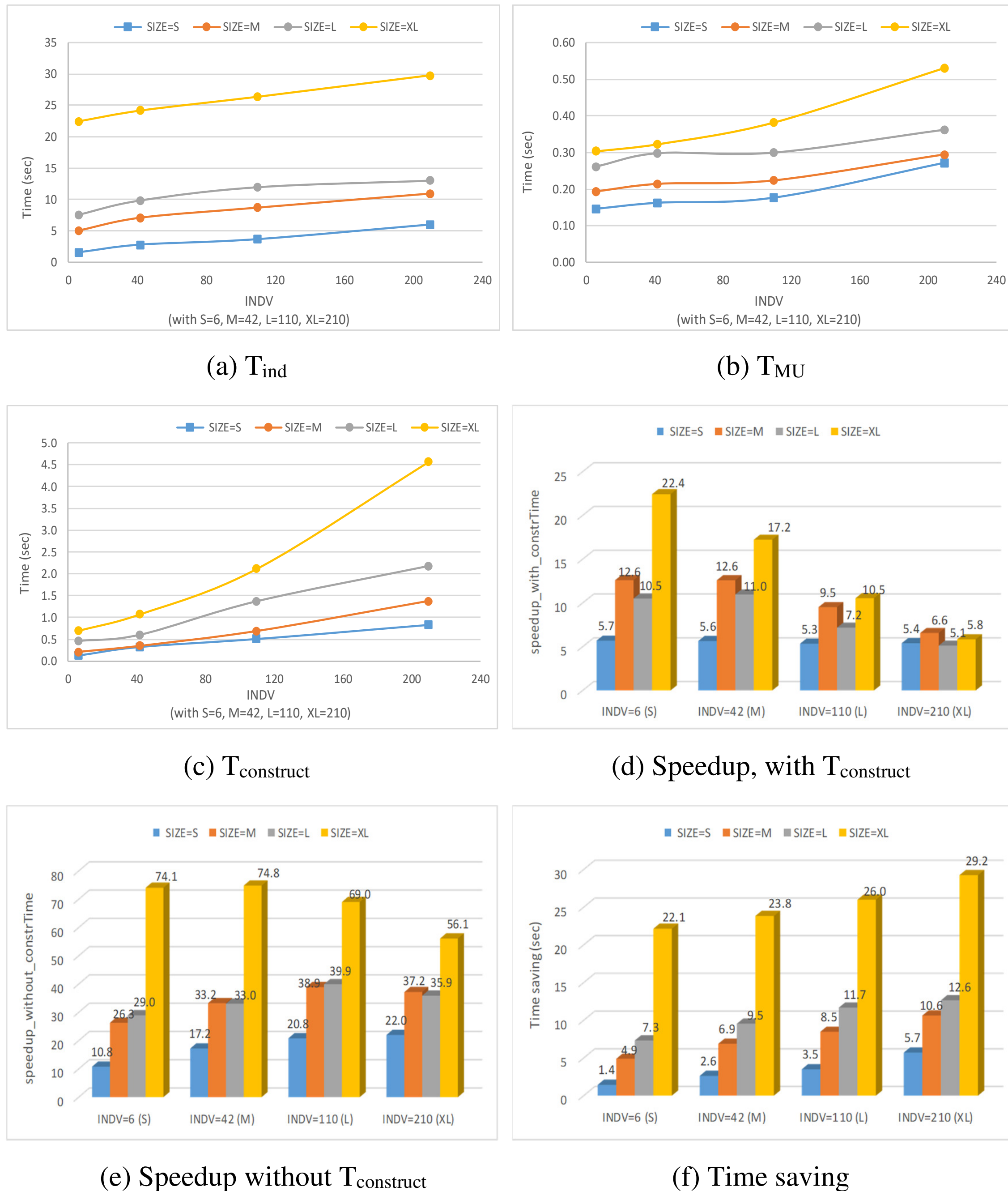
The *TimeSaving* metric, however, can be relied on to observe the behavior of the time gained from using M_U to perform a particular task as opposed to using individual models, multiple time. Figure 57(f) clearly illustrates a consistent pattern of time savings, which increases when *SIZE* and *INDV* increase, and this is actually the result that we were hoping for.

5.4. Results for Trend Analysis (RT2)

In this experiment, a trend analysis is conducted on an element named X-Goal from a set of individual GRL models and their union model M_U. The purpose of this analysis is to study the trend of this goal's *importance value* attribute and analyze how this value changes over time. Performing this analysis on M_U implies retrieving an element named X of type Goal, annotated with any version number `<versions>`[10], which could be a single version, a set of versions, or a range of versions that the element may belong to. With individual models, the search for and retrieval of X-Goal involve each individual model, where the (laborious) process in practice would also involve opening each individual model, searching the desired element, observing its importance value, and closing the current model.

Figure 58(a)-(c) respectively show the T_{ind}, T_{MU}, and $T_{construct}$ utilised, while Figure 58(d)-(f) illustrate the time speedups, with and without $T_{construct}$, and the time saved in this experiment. The results in Figure 58(a)-(c) clearly illustrate a linear increase of T_{ind}, T_{MU}, and $T_{construct}$ as *SIZE* or *INDV* increases. This is expected as the searching task (which is the core of trend analysis) has a linear time complexity.

[10] See STAL grammar in Appendix B



(a) T_{ind} (b) T_{MU}

(c) $T_{construct}$ (d) Speedup, with $T_{construct}$

(e) Speedup without $T_{construct}$ (f) Time saving

Figure 58 Summary of results for trend analysis (RT2), for all INDV and SIZE categories

From Figure 58(d), it can be noticed that for one *SIZE* category (e.g., *SIZE*=XL), the speedup decreases with the increase of the number of individual models in a family (i.e., *INDV*). This decrease is mainly due to the consideration of $T_{construct}$ while computing the speedup. Nevertheless, the achieved speedups, with $T_{construct}$, are still very positive and

important. Without considering $T_{construct}$, the speedup becomes more substantial, with a generally increasing pattern as *INDV* or *SIZE* increases, as depicted in Figure 58(e).

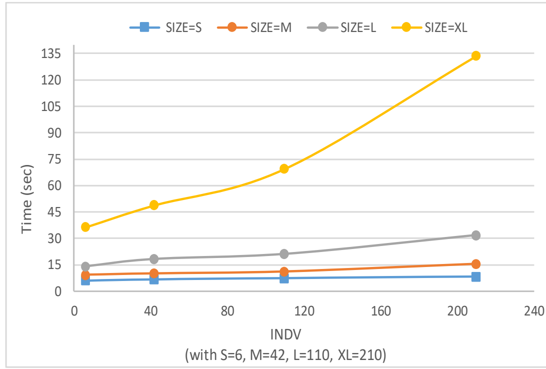
Finally, Figure 58(f) demonstrates that the use of M_U reduces the time needed to search for elements that belong to a group of models instead of traversing each individual model, separately. This is clearly illustrated by the time savings that substantially increase when *SIZE* or *INDV* increases.

5.5. Results for Commonality Analysis (RT3)

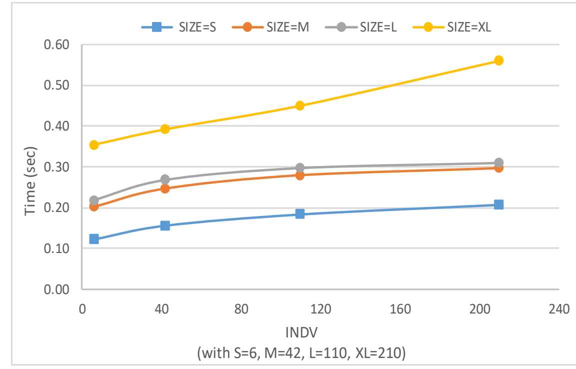
Figure 59 shows the results of conducting commonality analysis on a set of GRL models and their M_U. This experiment required searching for all elements that are common between all model versions. This is a tedious task, especially when the number/size of models increases. Searching a set of *M* individual models, with *N* elements each, to find elements in common between all models has a complexity of $O(M \times N^2)$. However, with M_U, we only use one model to search for elements in common, where the task here is to search for elements annotated with *<ALL>*.

It can be noted from Figure 59(a) that T_{ind} in this experiment is at least two times larger than T_{ind} in the previous experiment (i.e., RT2). T_{MU} and $T_{construct}$ on the other hand are growing with the same pace. Here again, the times T_{ind}, T_{MU}, and $T_{construct}$ increase as *SIZE* or *INDV* increases.

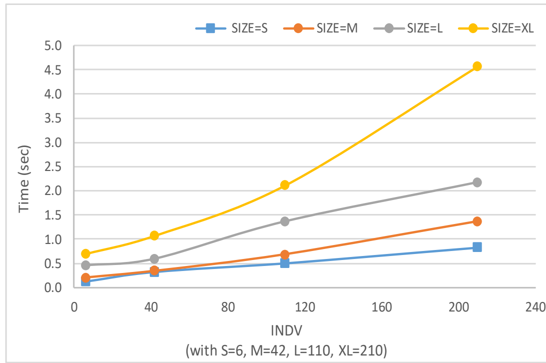
The time speedup achieved by using M_U is always positive, regardless of $T_{construct}$ being considered or not. Figure 59 (d) illustrates that the *speedup_with_constrTime* shows a pattern close to that of RT2. That is, for one *SIZE* category (e.g., *SIZE*=S or *SIZE*=M), the speedup decreases as *INDV* increase. The *speedup_without_constrTime*, on the other hand, is more substantial, and increases as *INDV* or *SIZE* increases, as shown in Figure 59 (e). Finally, the time savings in this experiment (Figure 59(d)) are more significant than in the experiments for RT2, as the potential gain here is quadratic rather than linear, with about 133 seconds saved for extra-large families of extra-large models.



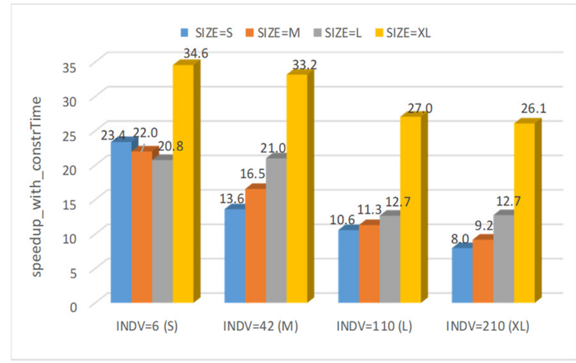
(a) T_{ind}



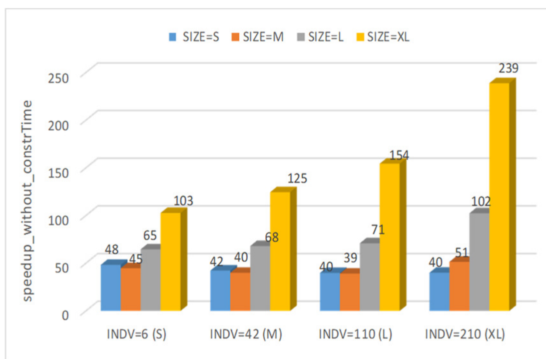
(b) T_{MU}



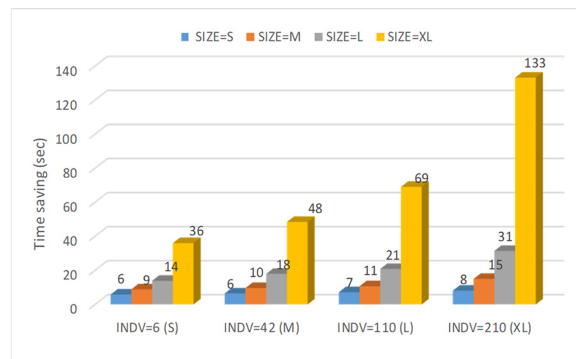
(c) $T_{construct}$



(d) Speedup, with $T_{construct}$



(e) Speedup without $T_{construct}$



(f) Time saving

Figure 59 Summary of results for commonality analysis (RT3), for all *INDV* and *SIZE* categories

5.6. Threats to Validity

One major threat to the validity of our empirical evaluation stems from relying on randomly generated inputs (both graphs and experimental parameters). This threat could be alleviated by using more realistic parameters, e.g., using real-world model families.

Another threat is related to the experimental parameters, where we used only *SIZE* and *INDV*. We recognize that we need to examine the impact of the variability of models on reasoning. For example, we could consider the number of different annotations per element to describe how similar or different the members are. The topology of the graphs (e.g., depth, number of linked nodes, etc.) could also benefit from specific experiments. The complexity of a property to be checked might also be another parameter to consider.

Our experiments need to be elaborated further for more complex properties and analysis types (some of which could perhaps not exploit the STAL annotations as the ones used here), and also be compared to approaches that handle some variability in the time dimension (only) for goal models, including the work of Aprajita et al. [93] and of Grubb and Chechik [208].

Furthermore, the current validation covers two modeling language (goal models and state machines) and it should be extended to other types that are more structural (e.g., class diagrams) or behavioral (e.g., process models). Finally, the usefulness of our approach needs to be assessed and demonstrated with more significant examples or real-world case studies.

5.7. Chapter Summary

This chapter explored RQ2 by evaluating, empirically, the efficiency of reasoning and analysis tasks using union models in comparison to the use of individual models. The chapter defined three general reasoning tasks and evaluated their performance using first, union models, and then using individual model several times. The chapter also discussed the experimental methodology, setup, and implementation and reported on the empirical results. Our experiments demonstrate the usefulness and performance gains of union models for analyzing a family of models, all at once, compared to individual models, with several threats to validity identified as a caveat.

The speedups, whether the time for constructing the union model is considered or not, are always in favor of the union model, and these speedups generally increase as models or families get larger. However, there was an erratic behaviour regarding the increase of the time speedup. For instance, the speedup for L-sized models was, in some experiments, larger than the speedup of M-sized models, and in other experiments it was smaller than that for M-sized models. This behaviour is due in part to the possibility of having a footprint for the creation of M_U, which is not negligible. Moreover, in order for the behaviour of the speedup to be more stable and less erratic, we may need to consider more experimental parameters, such as the ratio of variability across models, and the topology of the models at hand.

In addition, it is worthwhile to mention here that in some individual experiments related to the calculation of *speedup_with_constrTime*, we encountered some counter-intuitive results, where the speedup was less than one. In such experiments, a less-than-one speedup value suggests that the use of M_U, when taking its construction time into account, is slower than the use of individual models. Again, these slowdowns where obtained in individual experiments only, and they where amortized by calculating the average result of 15 runs.

In terms of concrete times saved, which range from a half-second to a few minutes depending on the experiment, they may not seem large at first glance but:

- They will accumulate as many analyses are performed on a same union model, especially as the union model construction time gets amortized;
- They still do not consider the concrete time required when a practitioner uses a tool to open a model, perform the analysis, and save the results (and this may take many seconds per model, in addition to being prone to human errors);
- They become essential in a context where union model construction and verification are offered *as a service*, i.e., as an online application where multiple users can concurrently upload, merge, and analyze their model families.

While this chapter focused on analysis types that are language *independent*, the next chapter explores the adaptation of language-*dependent* analyses for a given language, namely forward and backward propagation for GRL model families.

Chapter 6. Adapting Analysis Techniques for Goal Model Families

In the previous chapter, we investigated RQ2 by using *general* reasoning tasks that can be applied to any modeling language. This chapter continues the exploration of a related research question, but this time with analysis techniques that are *specific* to one modeling language. In particular, we target *goal modeling languages*, with GRL as an example, and we explore **RQ3** in that context: *Can there be a performance gain from adapting existing analysis techniques specific to the GRL language on a family of GRL models, all at once using M_U, compared to analyzing individual GRL models, one model at a time?*

This chapter aims to answer RQ3, and refines it as follows: if *A* is the existing analysis algorithm applied to individual GRL models, and *A'* is the adapted analysis algorithm to be applied on M_U, and if *P* is the performance (in terms of runtime) of applying *A* on individual models, one model at a time, and *P'* is the performance of applying *A'* on M_U, then we need to examine if there is any performance gain (G) defined as $G=P/P'$. A G value greater than 1 represents a gain whereas a G value between 0 and 1 is a loss.

The reason why we select GRL as a target goal-oriented language instead of *i** [210], the NFR framework [211], Tropos [212], or KAOS [213] is because GRL has a clear *arithmetic semantics* defined formally by Fan et al. [209]. This semantics can be adopted and adapted (or lifted) for GRL model families.

The rest of this chapter is organized as follows: Section 6.1 briefly introduces GRL, together with its syntax and selected analysis techniques. Section 6.2 discusses the arithmetic semantics of the standard GRL and its forward propagation algorithm and discusses how to adapt it for GRL model families. The standard backward propagation algorithm and its adaptation are discussed in Section 6.3. An illustrative case study is discussed in Section 6.4. Our empirical results are discussed in Section 6.5. Finally, Section 6.6 summarizes the chapter.

6.1. Goal-oriented Requirement Language

GRL is a graphical goal modeling language rooted in the *i** [210] and Non-Functional Requirements (NFR) [211] frameworks. GRL consists of concepts inspired from both frameworks such as intentional elements, actors, and links (from *i**), as well as evaluation mechanisms (from the NFR framework). GRL helps capture intentions, business goals, and non-functional requirements of systems and stakeholders, together with their relationships [7]. The subsequent sections briefly discuss GRL's syntax and the propagation analysis techniques used to evaluate GRL models.

6.1.1 GRL Syntax

A GRL goal model is an acyclic directed graph that consists of *intentional elements* connected by *intentional links*, where intentional elements and links may optionally reside within an *actor*.

- Intentional elements include goals, softgoals, tasks, and resources.
- Intentional links include decomposition (of type AND, OR, or XOR), contribution (qualitative or quantitative), and dependency links.
- Actors are the active entities (e.g., stakeholders or systems themselves) who want goals to be achieved, softgoals to be satisfied, tasks to be performed, and resources to be available.

A detailed summary of the standard GRL syntax is presented in Appendix A.

A GRL *strategy* describes a particular configuration of alternatives in the GRL model, by assigning an initial qualitative value or a quantitative evaluation value (e.g., a value between 0 and 100) to some of the intentional elements in the model. A GRL *evaluation mechanism* then propagates evaluation values to other high-level intentional elements, through intentional links, and computes their satisfaction values as well as the satisfaction of actors and of the entire model.

6.1.2 Propagation-based Analysis in GRL

Different evaluation algorithms exist for GRL, including qualitative, quantitative, and hybrid propagations that are either *bottom-up* or *top-down*. In this thesis, we consider both

categories of propagation-based quantitative analysis techniques available for standard GRL models, and we adapt/lift them to the context of GRL model families.

Generally, the bottom-up analysis algorithms aim to answer what-if questions such as "*what is the impact of this set of choices*". These algorithms start with assigning initial values to leaf elements of the model (a strategy that usually represents choices between alternative solutions), and then use model links to propagate values upward to higher-level elements. An example of this category in the GRL is the *forward propagation analysis* algorithm proposed by Amyot et al. [214]. The top-down analysis algorithms, on the other hand, aim to answer questions such as "*what alternatives and values in the leaf elements will achieve this top-level goal?*". These algorithms start by choosing a top-level goal to satisfy, and then search for the combination of solutions that best meets that goal. The *backward propagation algorithm* in GRL [211] is an example of this category. Forward propagation has a complexity similar to testing and is hence very fast, whereas backward propagation is slower with a much higher complexity, similar to search-based algorithms.

The subsequent sections elaborate further on these two analysis algorithms and discuss their original semantics as well as the adapted semantics for the model family context.

6.2. Forward Propagation Analysis in Standard GRL

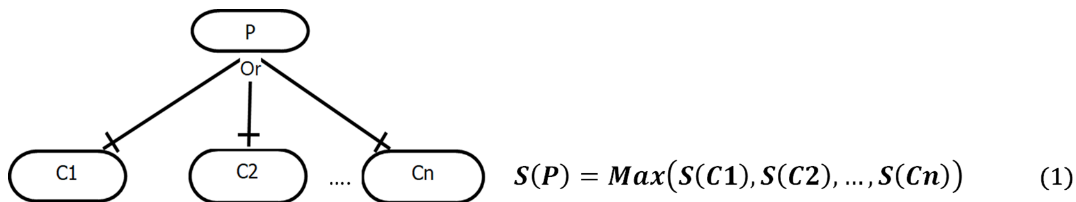
This section discusses the arithmetic semantics of GRL's forward propagation algorithm, and the adapted semantics used for GRL model families.

6.2.1 Arithmetic Semantics of GRL Forward Propagation Algorithm

In a GRL goal graph, intentional elements *IE* have a satisfaction value *S(IE)* computed based on a selected GRL strategy or external inputs. IEs could also have an importance value, *imp(IE)*, representing their weight in the computation of the satisfaction of their containing actor, if any [209]. In addition, contribution links could be *weighted*, where the weight value of a contribution link *w(L)* represents the impact of that link on the satisfaction of its source element. Furthermore, an actor *A* (as in the case of *IE*s) could also have a satisfaction value *S(A),* as well as an importance value *imp(A)* used in the computation of the satisfaction of the entire model [209].

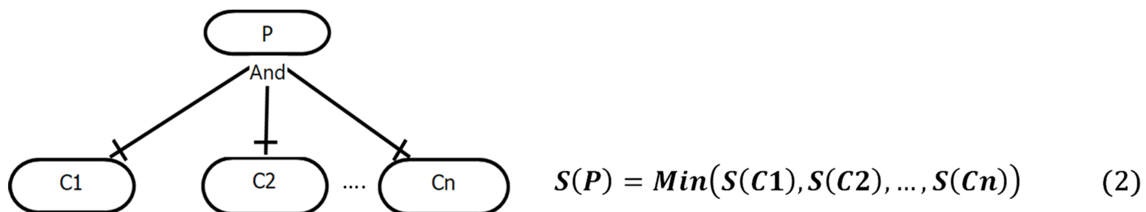
The forward propagation of initial satisfaction values towards higher-level goals depends on the type of the intentional links (*IL),* through which satisfaction values are propagated from a child element *C* into a parent element *P*. An *IL* is a *decomposition,* a *contribution*, or a *dependency link*. For an *OR-type* decomposition link, the satisfaction level of *P* is the *maximum* value of the quantitative evaluation of its *Ci* elements. For an *XOR-type* decomposition link the *maximum* is also used, but a warning is generated if more than one child element has a quantitative evaluation value different from 0 [6]. Finally, the satisfaction level of an element with an *AND-type* decomposition link is the *minimum* value of the quantitative evaluation values of its source elements, *Ci* [6]. In case of *qualitative evaluations*, the ordering of evaluations is *satisfied*> *weakly satisfied*> *weakly denied* >*denied*> *conflict*>*unknown*>*none*. However, in this thesis, we only consider *quantitative evaluations* of elements, where contribution values are between –100 and +100, and satisfaction and importance values are between 0 and 100.

Figure 60 and Figure 61 show examples of an OR-type decomposition, and an AND-type decomposition, respectively, along with their arithmetic semantics adopted from Fan et al. [209].



$$S(P) = Max\big(S(C1), S(C2), \dots, S(Cn)\big) \qquad (1)$$

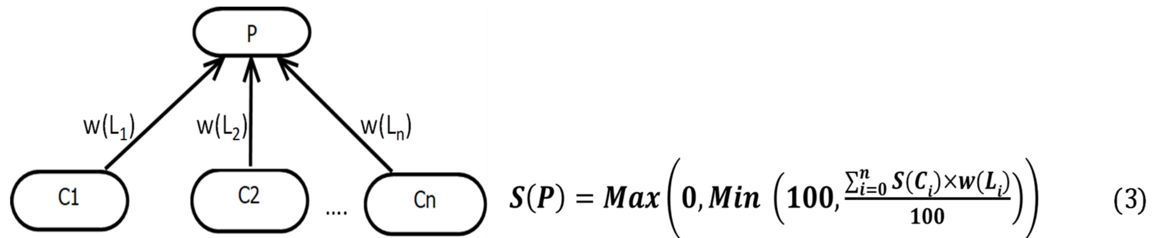
Figure 60 OR-type decomposition links and their arithmetic semantics



$$S(P) = Min\big(S(C1), S(C2), \dots, S(Cn)\big) \qquad (2)$$

Figure 61 AND-type decomposition links and their arithmetic semantics

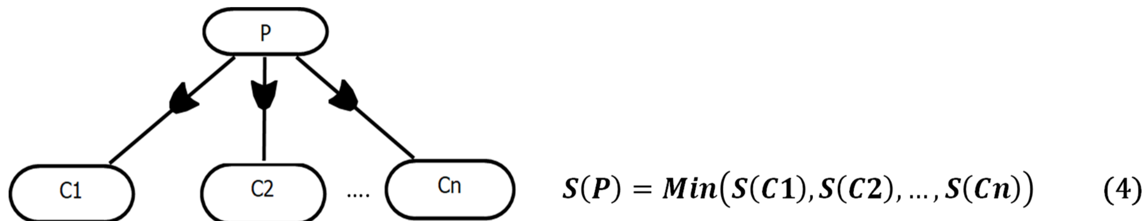
Contribution links propagate the evaluation values as the truncated sum of the products (i.e., weighted sum) of the quantitative evaluation of each source element $S(C_i)$ multiplied by its quantitative contribution level or weight, $w(L_i)$, to the parent element. Figure 62 illustrates an example of (weighted) contribution links and their arithmetic semantics.



$$S(P) = Max\left(0, Min\left(100, \frac{\sum_{i=0}^{n} S(C_i) \times w(L_i)}{100}\right)\right) \quad (3)$$

Figure 62 Contribution links and their arithmetic semantics

For *dependency links*, the source element of such links, *P*, cannot have an evaluation value higher than those of the containable elements it depends on (i.e., the target elements of the dependency links). The forward propagation algorithm hence simply takes the minimum between the evaluation values of the target elements, $S(C_i)$, as illustrated in Figure 63.



$$S(P) = Min(S(C1), S(C2), \ldots, S(Cn)) \quad (4)$$

Figure 63 Dependency links and their arithmetic semantics

In a GRL graph, any intentional element could have multiple types of links simultaneously. For instance, an element *P* can be connected to elements C_i by an OR-decomposition links, a contribution link, and a dependency link, as shown in Figure 64. In this case, and as specified in the URN standard [6], the quantitative evaluation of decomposition links is computed first, then the evaluation of contribution links are added, and finally the evaluation of dependency links is added. Figure 64 shows the arithmetic semantics of the forward propagation algorithm with a combination of links.

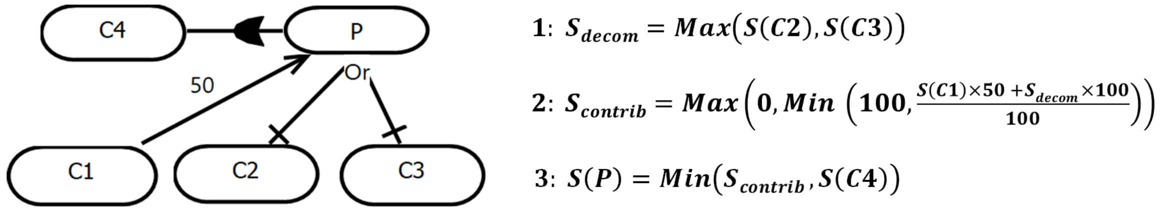


Figure 64 Multiple types of links and the arithmetic semantics of their propagation

To compute the quantitative evaluation value of an *actor* (i.e., *S(A)*), it is necessary to first identify the satisfaction value, *S(IL),* and the importance value *imp(IL)* of each contained element bound to that actor (see Figure 65). Only elements with an *imp(IL)* greater than 0 are counted. In addition, a truncated weighted average is used if the sum of the importance values is greater than 100, otherwise a truncated weighted sum is used [209].

Finally, the quantitative evaluation value of the entire *GRL model M* is calculated the same way as for actors, except that the quantitative evaluation values and quantitative importance values of actors (i.e., *S(A)* and *imp(A)*) are used here instead of containable intentional elements, as illustrated in Figure 66.

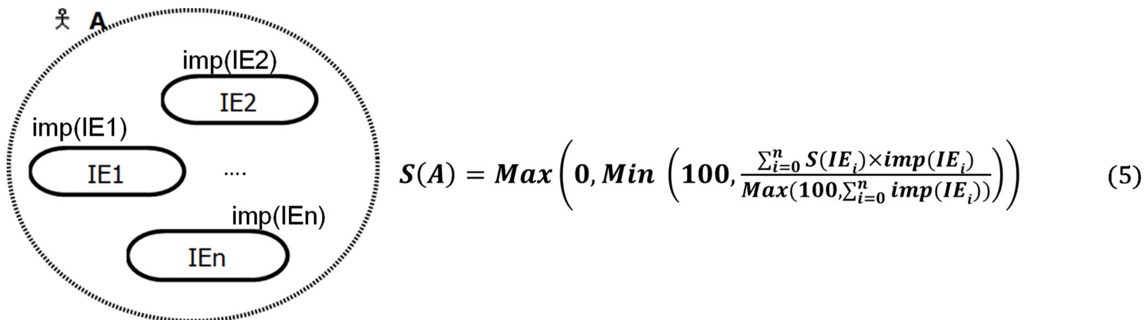
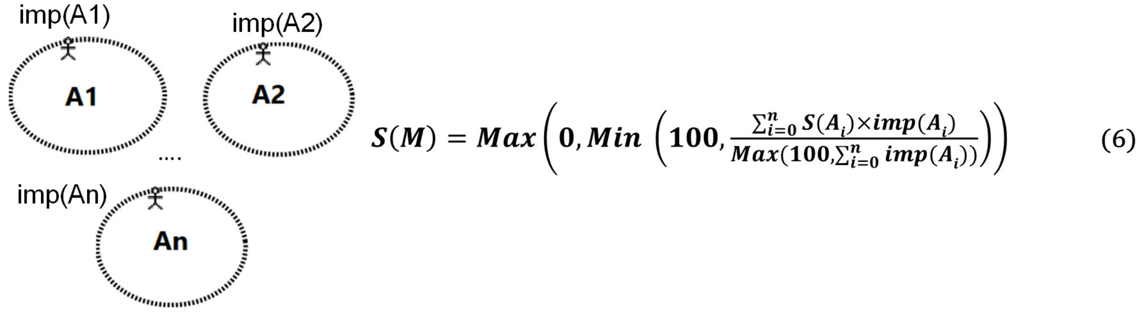


Figure 65 An actor containing multiple IEs with their importance values, and its arithmetic semantics



$$S(M) = Max\left(0, Min\left(100, \frac{\sum_{i=0}^{n} S(A_i) \times imp(A_i)}{Max(100, \sum_{i=0}^{n} imp(A_i))}\right)\right) \quad (6)$$

Figure 66 A GRL model with multiple actors, and the arithmetic semantics of M's quantitative evaluation

6.2.2 Adapted Semantics for GRL Model Families

In this section, we *adapt* the arithmetic semantics of GRL's forward propagation algorithm discussed above to fit with the nature of the proposed union model, M_{U}. In particular, the calculation of the quantitative evaluations of *IEs*, actors, or the entire model (M_U in this case) becomes aware of the evolution/variation of models by taking into consideration the annotations of model elements. Hence, for the same M_U, there will be multiple quantitative evaluation values, depending on the annotations of M_U's elements. In other words, the arithmetic functions illustrated in equations 1-6 for GRL model components (i.e., OR and AND decomposition links, contribution links, dependency links, actors, or the entire model) are used as is in the lifted or adapted propagation algorithm for model families, but this time, with each function having a list of evaluations (i.e., a *multi-valued function*), depending on the annotations attached to each component.

Figure 67(a)-(c) show an example of a model that evolves over time into three versions (ver_1-ver_3) and their M_U (Figure 67(d)). The three models are assumed to belong to the same configuration, say $Conf_A$, in the space dimension. According to the original arithmetic semantics, the GRL forward propagation algorithms for versions 1-3 are expressed as follows:

$$\boldsymbol{ver_1}: \mathrm{S(G1) = Min(S(G2), S(G3)\ )}$$

$$\boldsymbol{ver_2}: \mathrm{S(G1) = Max}\left(0, \mathrm{Min(100,} \frac{\mathrm{(\ Min\ (S(G2),S(G3)\ ) \times 100 + (}S(G5) \times 25) \times 40)}{100}\right)$$

$$\boldsymbol{ver_3}: \mathrm{S(G1) = Max}\left(0, \mathrm{Min\ (100,} \frac{\mathrm{(\ Min\ (S(G2),S(G3)\ ) \times 100 + (}S(G4) \times 50))}{100}\right)$$

The adapted arithmetic semantics for M_U uses the *piecewise* function to handle STAL annotation. The piecewise function uses a sequence of pairs (function, condition) that can be used to handle different cases corresponding to different annotations (annot). For Figure 67(d), this corresponds to S(G1) =

$$\mathrm{Max}\left(0, \mathrm{Min}(100, \frac{\left(\mathrm{Min}(\mathrm{S(G2)}, \mathrm{S(G3)}) \times 100 + \mathit{Piecewise}\left(\begin{matrix}((S(\mathrm{G5}) \times 25) \times 40, \mathrm{annot}{=}{=}\mathrm{ver2}), \\ (S(\mathrm{G4}) \times 50, \mathrm{annot}{=}{=}\mathrm{ver3})\end{matrix}\right)\right)}{100})\right)$$

which is equivalent to this multi-valued arithmetic equation:

$$S(\mathrm{G1}) = \mathrm{Max}\left(0, \mathrm{Min}\left(100, (\mathrm{Min}(\mathrm{S(G2)}, \mathrm{S(G3)}) \times 100 + \begin{cases} (S(\mathrm{G5}) \times 25) \times 40, & \text{if annot} = \boldsymbol{ver2} \\ S(\mathrm{G4}) \times 50, & \text{if annot} = \boldsymbol{ver3} \end{cases} /100)\right)\right)$$

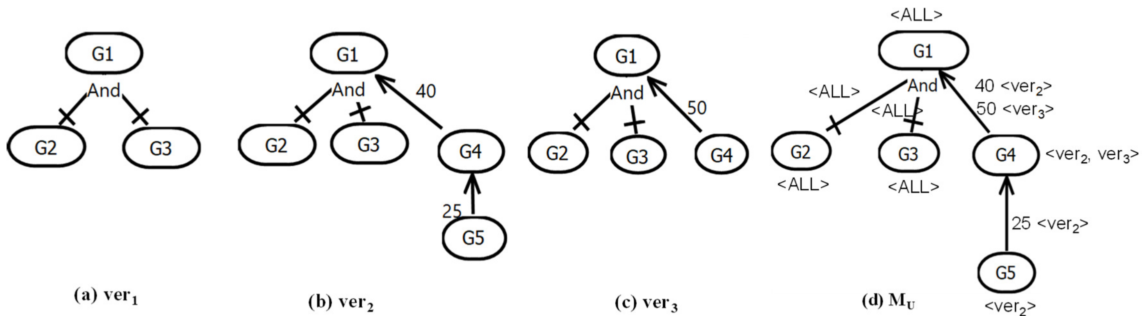


Figure 67 Examples of three model versions (a)-(c), and their union model MU (d)

As can be noticed, the usage of M_U here saved *redundant calculations* related to the AND decomposition part (i.e., $\mathrm{Min}(\mathrm{S(G2)}, \mathrm{S(G3)})$), which must be calculated otherwise with each individual model. The higher the degree of overlap between models in a family, the larger the models, and the larger the number of family members, the higher the computation savings.

6.3. Backward Propagation Analysis in Standard GRL

Backward analysis focuses on the *backward search* of all possible input values leading to some desired final objective, under desired constraints [215][216]. Given a GRL goal graph, the principle of backward propagation analysis of that graph is as follows: a modeler

decides what to maximize or minimize (often root goals, actors, or the entire model), then she searches, through forward propagations rules, for all possible initial assignments of input goals (often leaf goals) that would achieve the desired optimization. Multiple equivalent solutions may exist.

Having said that, it can be inferred that such a backward propagation algorithm analyses a goal model in a *brute force* manner. In general, there may be many satisfying assignments, up to exponentially many, corresponding to solutions for the problem [212][215][216]. To control the solution space, and to reduce conflicts, a modeler may enforce some *constraints* on the solutions to be returned.

6.3.1 Semantics of GRL Backward Propagation Algorithm

Given the arithmetic semantics of GRL's forward propagation algorithm (Section 6.2.1), we reduce the problem of backward propagation in GRL into that of *constrained optimization* [217], with the arithmetic functions of root-level *IE*s being the *objective functions* to be achieved (i.e., maximized or minimized), subject to some *constraints* on the evaluation values of (often leaf-level) IEs, where leaf-level IEs are considered as *decision variables*. This approach was first proposed by Anda and Amyot [218].

In our context, we consider the objective functions (i.e., the quantitative satisfaction levels of IE, A, or M) as a reward or a utility function which is to be *maximized*. Hence, the constrained optimization problem becomes simply a *constrained maximization*. For instance, given an objective function $f(x)= S(IE_i)$, and a constraint function $g(x)=const$, then the generic format of the constrained maximization is expressed as:

$$\begin{array}{ll} Maximize & f(x) = S(IE_i) \\ subject\ to & g(x) = const \end{array}$$

To achieve this, we use a constraint programming (CP) optimizer implemented in the IBM ILOG CPLEX Optimization Studio [219]. This optimizer applies linear programming (LP) or constraint programming [220] to analyze candidate solutions and select the optimal ones. The use of CPLEX for optimizing GRL models was proposed by Anda and Amyot [218].

For example, given the goal model ver_2 in Figure 67 (b), if we want to maximize the quantitative evaluation of the root goal G1, with a constraint that the leaf goal G5 is

initially assigned a quantitative evaluation equal to 25, then this can be expressed as follows:

$$\boldsymbol{Maximize}\ \mathrm{S(G1)} = \mathrm{Max}\left(0, \mathrm{Min}(100, \frac{(\mathrm{Min}\ (\mathrm{S(G2)}, \mathrm{S(G3)})\ ) \times 100\ + (S(G5) \times 25) \times 40)}{100}\right)$$

$$\boldsymbol{Subject\ to}\ \mathrm{G5} = 25$$

In CPLEX, the above example is expressed in Listing 2 as follows:

```
//Model ver2 at Figure 67 (b):
using CP;

//Decision variables that can be constrained
dvar int G2 in 0..100;
dvar int G3 in 0..100;
dvar int G5 in 0..100;

//Arithmetic function of goal G1
dexpr float G1=
maxl(0.0,minl(100.0,(40*maxl(0.0,minl(100.0,(25*G5)/100.0))
                       +minl(G2,G3)*100.0)/100.0));

//Objective function: to maximize the satisfaction level of goal G1
given an initial satisfaction of G5=25

maximize G1;
subject to {G5==25;}
```

Listing 2 CPLEX code for model "ver2" in Figure 67 (b)

In this example, G2, G3, and G5 are decision variables (`dvar` in CPLEX) from which the CP optimizer will find an optimal solution (e.g., G1 = 100). Constraints can be enforced on zero, one, or more of these decision variables.

The CPLEX representation of the individual goal models ver_1 and ver_3 (Figure 67 (a) and (c)) is illustrated in Listings 3 and 4, respectively (where the objective here is to maximize the quantitative evaluation of the root goal G1).

```
//Model ver1 from Figure 67 (a):
using CP;

//Decision variables that can be constrained
dvar int G2 in 0..100;
dvar int G3 in 0..100;

//Arithmetic function of goal G1
dexpr float G1= minl(G2,G3);

//Objective function:
maximize G1;
```

Listing 3 CPLEX code for model "ver1" in Figure 67 (a)

```
//Model ver3 from Figure 67 (c):
using CP;

//Decision variables that can be constrained
dvar int G2 in 0..100;
dvar int G3 in 0..100;
dvar int G4 in 0..100;

//Arithmetic function of goal G1
dexpr float G1 =maxl(0.0,minl(100.0,(50*G4 + minl(G2,G3)*100.0)/100.0));

//Objective function:
maximize G1;
```

Listing 4 CPLEX code for model "ver3" in Figure 67 (c)

6.3.2 Adapted Semantics for GRL Model Families

To adapt backward propagation for a GRL model family, we follow the semantics of the standard backward propagation discussed above in Section 6.3.1, except that we use here the adapted, annotation-aware forward propagation rules (Section 6.2.2) to determine which quantitative evaluations of leaf-level *IE*s, along with their different annotations, achieve the desired optimization. Since the adapted forward propagation rules are used here as well, then each element in a GRL model may have multiple arithmetic functions, with different evaluations, depending on the annotations of that element, as already discussed in Section 6.2.2.

To this end, the different evaluations of the same element can be considered as a potential initial assignments of input goals that would achieve the desired optimization objective. Hence, the solution space would involve all possible assignments for all elements and their variations. For example, given the union model M_U in Figure 67 (d),

maximizing the quantitative evaluation of the root goal G1, such that the leaf goal G5 is initially assigned to 25, can be expressed as:

$$\textbf{\textit{Maximize}}\ \mathrm{S(G1)} = \ \mathrm{Max}\left(0, \mathrm{Min}\left(100, \frac{\left(\mathrm{Min}(\mathrm{S(G2)},\mathrm{S(G3)})\times 100 + \textit{Piecewise}\left(\begin{matrix}((S(\mathrm{G5})\times 25)\times 40, \mathrm{annot}{=}{=}\mathrm{ver2}), \\ (S(\mathrm{G4})\times 50, \mathrm{annot}{=}{=}\mathrm{ver3})\end{matrix}\right)\right)}{100}\right)\right)$$

$$\textbf{\textit{Subject to}}\ \mathrm{G5} = 25$$

In this example, to calculate the maximum satisfaction of G1, we consider both annotations on the contribution link between G4 and G1. Hence, the evaluation of G1 changes, based on the different annotations of that contribution link (or other elements) that directly affect G1. In CPLEX, the above example is expressed as shown in Listing 5:

```
//Union model MU from Figure 67 (d)
using CP;

//decision variables
dvar int G2 in 0..100;
dvar int G3 in 0..100;
dvar int G4 in 0..100;
dvar int G5 in 0..100;

//decision variables for version/configuration annotations
dvar int ver in 1..3;
dvar int conf in 1..10; // equivalent to confA.. confJ

//Objective function: to maximize the satisfaction level of goal G1
//given an initial satisfaction of G5=25

dexpr float G1= maxl(0.0,minl(100.0,
       (ver==2? 40*maxl(0.0,minl(100.0,(25*G5)/100.0)/100.0):
              (ver==3? maxl(0.0,minl(100.0,(50*G4)/100.0)):
                     0
              )
       ) +
       (minl(G2,G3)*100.0)/100.0
 ));

maximize G1;
subject to {G5==25;}
```

Listing 5 CPLEX code for the union model M_U in Figure 67 (d)

If we execute the code above, the optimizer returns a solution where G1=100 for ver=1 when G2=100 and G3=100, as illustrated in the “Solutions” tab in Figure 68. Indeed, this corresponds to a valid result for the model family in Figure 67. If we further constrain the

solution space with G2=50, then the optimizer correctly returns a solution where G1=100 for ver=3 when G3=50 and G4=100, as illustrated in the "Solutions" tab in Figure 69, which also corresponds to a valid result for the model family in Figure 67.

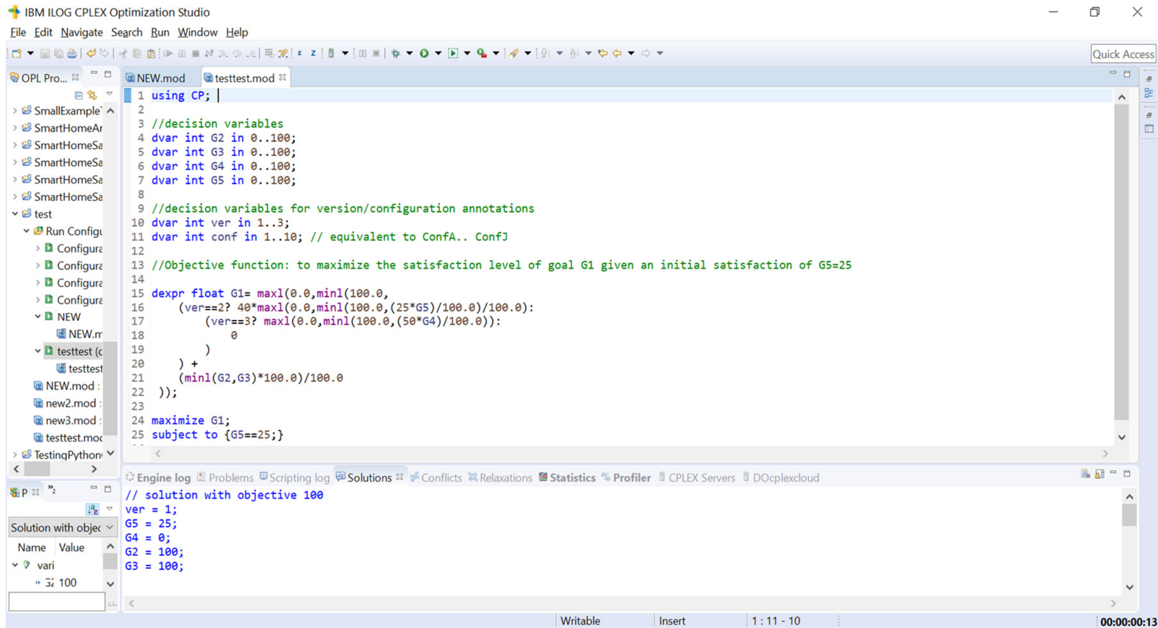


Figure 68 A solution for MU when G5=25

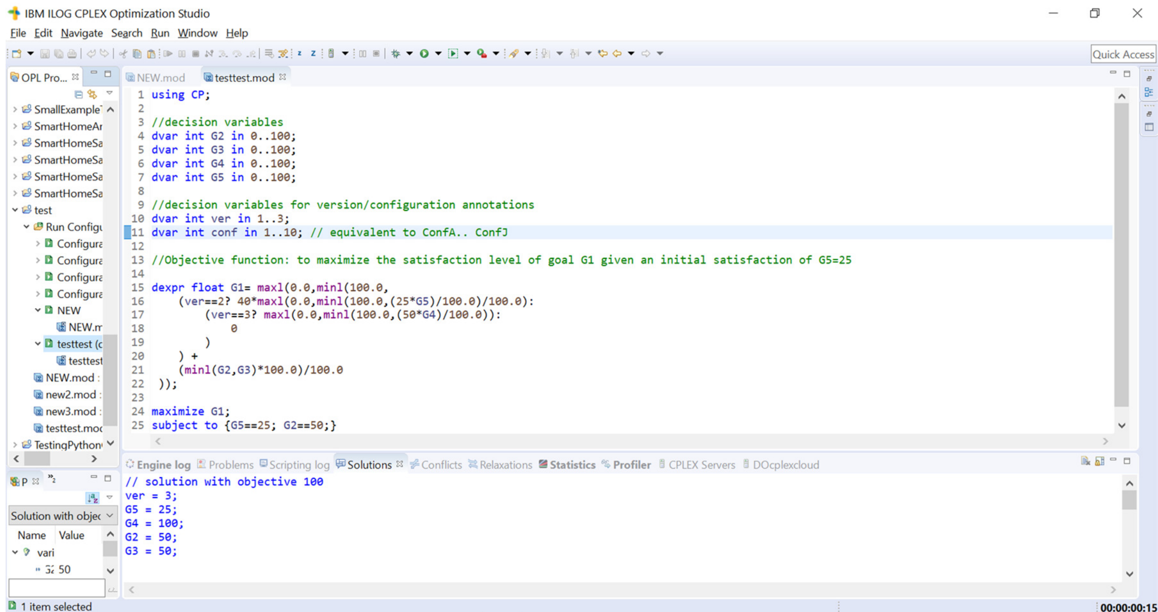


Figure 69 A solution for MU when G5=25 and G2=50

It is worth mentioning here that "`ver==2`" and "`ver==3`" used in the CPLEX example above are used as labels to indicate ver_2 and ver_3 respectively. When both space and time dimensions are used, one variable per dimension is needed, for example (`ver==2 && conf==1`).

6.4. Illustrative Case Study

In Chapter 1, I introduced an example from the regulatory domain (from Transport Canada) to motivate my approach. In this section, I used another example with a similar level of complexity of models and model families, namely a simplified smart home environment, published in [189]. The results from this section would very likely apply to the regulatory domain as well.

In this example, we have a GRL model family, where stakeholders' goals, importance of goals, means to achieve goals, and relationships between goals (e.g., contributions or decompositions) vary. Such variation stems mainly from the existence of different configurations of a smart home that also evolve over time. In this example, we distinguish between two different configurations:

- Configuration A (*confA*): A smart home that is lived in by students who spend at least 8 hours of the day out of home.
- Configuration B (*confB*): A smart home that is lived in by retired senior persons (most likely sick), who spend most of their daytime and nighttime at home.

In these two configurations (see Figure 70), a student's goals are slightly different from a senior's goals. For example, a student is more concerned about getting fresh air in her room by opening windows so as to reduce energy consumption (since a student's budget is usually tight). A retired senior, on the other hand, may focus more on getting her room's atmosphere refreshed using the most convenient option (regardless of cost), e.g., by having the ventilator turned on most of the time. Also, the importance of achieving the "refresh air inside" goal differs between a senior person (100) and a student (80). Furthermore, to keep a senior's smart home secure, the home central operator could give the illusion that the house is lived in using several options, one of them being to keep lights always on. However, this may not be a feasible option for a student, since turning lights on all the time consumes energy beyond a student's budget affordability.

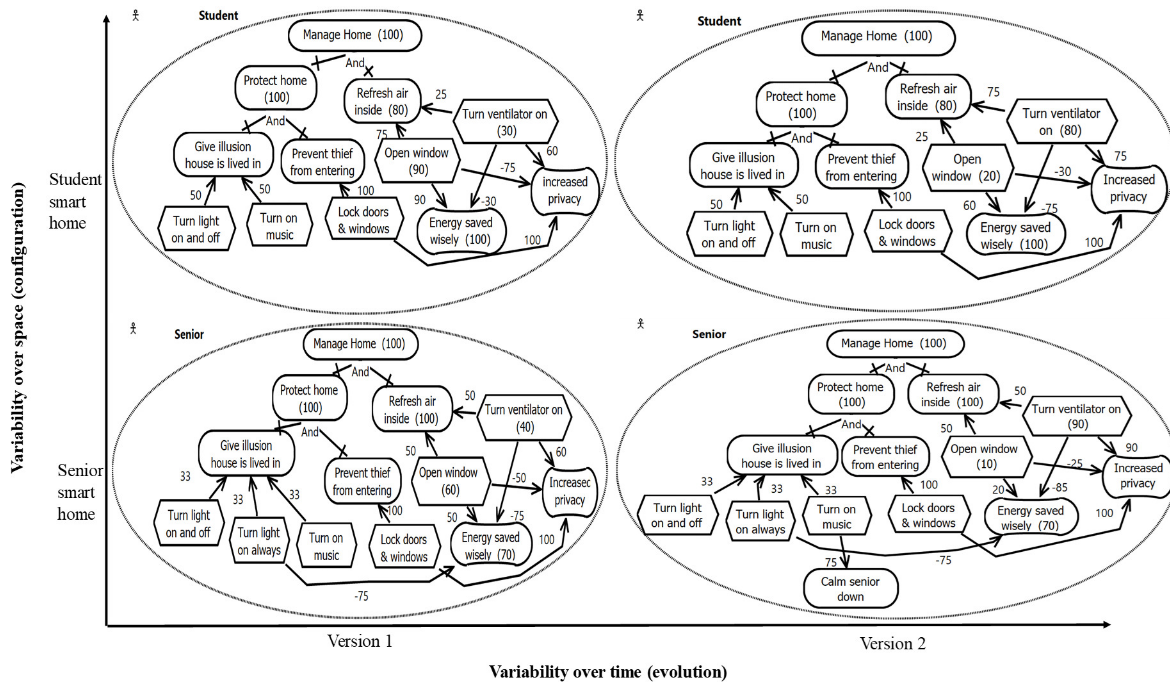


Figure 70 Goal model family for smart home environments varying according to space and time

In addition to these “space-based” variations, goal models (in both configurations) could also evolve over time. In this example, we illustrate the evolution of models after several months (however, evolution could also happen over shorter periods of time). In such time-based evolution, goals and the means to achieve them (i.e., tasks) may differ between version 1 (produced in the summer) and version 2 (produced in the winter), due to changes in temperature, humidity, daylight duration, etc.

For a student’s smart home (i.e., *confA*), the importance of goals/tasks and their impacts (i.e., contribution values) on other goals evolved from version 1 to version 2. For instance, the importance of task “Open window” in version 1 is 90 while it is 20 in version 2, as a student is able to open the window more often in the summer (assuming these models were produced in a Nordic country). Also, the impact of opening a window on the goal “energy saved wisely” is higher in the summer version (with contribution value=90) than in the winter version (60). Finally, opening a window often in the summer has a higher negative impact on the “increased privacy” softgoal.

The previous evolutions are also applicable in the senior smart home environment (i.e., *confB*). As Figure 70 shows, the “Open window” task is almost neglected at winter

time (with importance=10) compared to summer time (importance=60). This is because a senior person is more vulnerable to get cold in the winter. Also, in version 2 (winter), the possibility for seniors to get depressed and anxious is higher (due to snow fall and short daylight duration). In this version, a smart home operator may calm the senior down by turning on soft music.

If we want to analyze each member of the model family illustrated above, and search for all solutions that will maximize the satisfaction of the overall goal in the smart home system (i.e., to "Manage Home"), then we need to perform backward propagation analysis on each model, one model at a time. One important **challenge** that we will face here is related to the complexity and effort required to analyze such family of models (note that past versions may also require analysis in case old versions of a system are still used by customers). In the example illustrated in Figure 70, we would end up running the backward propagation algorithm four times, even though there are *many* common elements and computations among the four models. As mentioned previously, if there are M individual models in a model family, and each model has E elements, then the complexity of running a satisfaction propagation algorithm on all models would be in order of $M \times O(E)$. Such complexity becomes more significant if there are hundreds of models (or more), with hundreds of elements (or more) in each model. Moreover, the effort of loading a model into a tool, analyzing the model, saving analysis results, and then moving to the next model is *not negligible* in practice.

In the next section, we use the individual GRL models (Figure 70) of this case study as a starting point to conduct our empirical validation, and we grow these models to obtain model families of different sizes. In addition, to increase the solution space, we assumed that the satisfaction values of models' elements range from 1 to 100.

6.5. Experiments

This section describes our experiments to evaluate the performance (P) of applying the backward propagation algorithm, realized as a constrained maximization problem, on a set of individual GRL models, and compare it to the performance (P') of applying the adapted backward propagation on a family of goal models.

6.5.1 Methodology

As mentioned Chapter 5, we generate random but realistic GRL models as attributed typed graphs according to Def. 5. We developed a method for transforming entire GRL models into mathematical functions according to the arithmetic semantics of GRL proposed by Fan et al. [209]. The transformation of GRL models into arithmetic functions (equations 1-6 in Section 6.2.1) is implemented in Python 3.6 and exploits the arithmetic representation in the Optimization Programming Language (OPL). We used OPL as a target programming language because it is the language supported by the IBM ILOG CPLEX Optimization Studio [219]. The CPLEX-OPL code generated for the GRL models in Figure 70 is shown in Appendix C.

The generated mathematical functions are then taken as input objective functions for the CPLEX's CP optimizer. The CP optimizer covers a wide range of mathematical operators, including divisions and multiplications, used to express the arithmetic semantics of models. In addition, the CP optimizer supports discrete integer decision variables (with different values) for which the optimizer solves the constrained maximization problem.

For this performance study, I created different model families by taking the GRL models of the smart home system (Figure 70) as a basis, and then I enlarged them multiple times, with likely changes, to produce close-to-real models of different sizes. Model families are then created so as to have different numbers of models, where each individual model also varies in the number of elements it contains. To achieve this, I reused the experimental parameters from Chapter 5, namely *SIZE* (i.e., number of models/family) and *INDV* (i.e., number of elements per model), and I followed the same methodology to discretize the parameters' domains into categories. The categories of *SIZE* and *INDV* parameters are repeated here again (Table 4 and Table 5) for convenience.

Table 4 Categories of parameter *SIZE* (number of elements in a model)

#elements/model (*SIZE*)	(0, 30]	(30, 110]	(110, 420]	(420, 1640]
Exemplar	12	56	240	930
Category	S	M	L	XL

Table 5 Categories of parameter *INDV* (number of models in a family)

#of individual models (*INDV*)	(0, 20]	(20, 72]	(72, 156]	(156, 272]
Exemplar	6	42	110	210
Category	S	M	L	XL

I conducted several experiments to examine the different combinations of parameters *SIZE* and *INDV*. I also ran the same experiment 10 times in order to avoid odd results, such that each point in the charts represents the average of 10 experiments. The next four sections report on our empirical results, where the results are organized according to the experimental parameter *SIZE*.

I also followed the same methodology discussed in Chapter 5 to measure the time speedup achieved by using M_U, with and without considering the time needed to build M_U (referred to as $T_{construct}$). These speedup metrics, respectively, are: *speedup_with_constrTime*= $T_{ind} / (T_{construct} + T_{MU})$, and *speedup_without_constrTime* = T_{ind} / T_{MU}. In addition, I measured the time saving, in seconds, achieved by using M_U to perform backward propagation, calculated as *TimeSaving*=$(T_{ind}–T_{MU})$.

It is worth mentioning that the time $T_{construct}$ considered in this chapter is the same as the one reported in Chapter 5 for RT2 and RT3, as we aimed to unify the same empirical settings related to the size and topology of the GRL models and their corresponding union models. As mentioned in Chapter 5, $T_{construct}$ is quite small and can be performed once before being amortized over multiple analyses.

During this performance study, we considered the typical case of model evolution/variation, where a model family members change slightly, resulting in their M_U being quite larger than any one of the individual models. We avoid extreme cases where individual models either do not change at all (resulting in M_U being the same as any one of the individual models), or individual models change entirely (resulting in the M_U containing all elements of all models). With the typical scenario of model families, it is intuitively expected that the memory needed to process a union model should be larger than the memory needed to process any of its constituting individual models (particularly, the largest individual model). To verify this assumption, we considered the "Total Memory Usage" metric reported in the CP optimizer engine log and displayed at the CPLEX Studio IDE.

This metric represents the memory used by the IBM ILOG Concert Technology and the IBM ILOG CP Optimizer engine.

In particular, we recorded the memory usage of M_U of different *SIZE* and *INDV* categories, denoted to as *Memory*M_U, and also recorded the memory usage of each of the individual models in any model family, referred to as *MemoryIndv*. Then we calculated the percentage of the memory consumed by M_U as: *Memory consumption*= (*Memory*M_U – *Max(MemoryIndv))* / *MemoryIndv* × 100%, where *Max(MemoryIndv)* is the maximum amount of memory used for the largest individual model in a given family.

Although this thesis focuses primarily on the runtime as a metric to measure the performance gain achieved by using M_U; this chapter reports on the percentage of memory consumption by M_U for two reasons:

1) To compare between the amount of memory needed to process the largest individual model in a given model family against the memory needed to process the union model of that model family. For M_U usage to be feasible, the extra memory used to process M_U should not exceed the total memory needed for all individual models constituting that M_U. In other words, assume that model *A* is the largest individual model in a model family that needed an amount of memory equals to α. Assume also that another *N* models were added to the same model family. In this case, M_U (which captures model *A* and each of the *N* models) would be considered feasible as long as the memory consumed to process M_U is a few times larger than α, but not *N* times larger.
2) To be able to trade-off between time speedup versus memory consumption. In other words, to assess whether it is worthy to use M_U (as opposed to individual models) for analysis purposes given the time speedup M_U achieves and the memory it consumes. This trade-off analyses would be of particular importance when our approach is used in applications where memory is a critical resource (e.g., in cloud-based applications, where memory needs to be allocated dynamically), or in real-time applications where time is crucial.

I conducted several experiments to examine the different combinations of parameters *SIZE* and *INDV*. I also ran the same experiment 10 times in order to avoid odd results, such that

each point in the charts represents the average of 10 experiments. The next four sections report on our empirical results, where the results are organized according to the experimental parameter *SIZE*.

To build confidence about the empirical results, output solutions retrieved from running constrained optimizations on individual models, of a particular family, were compared to the optimization solutions returned for the corresponding union model, where annotations identify individual models in the union model. No problems were found.

All experiments were executed on a laptop with an Intel Core i5-8250U (8th Gen) 1.6 GHz quad-core CPU and 8GB RAM, running Windows 10-x64.

6.5.2 Results for SIZE=S and INDV=(S, M, L, XL)

This section discusses the results of conducting constrained maximization evaluation on a set of goal models of *SIZE*=S, where the number of elements per model ranges from 1 to 30 elements. Figure 71 shows that with *SIZE*=S and *INDV*=S, the time, in seconds, it took the CP optimizer to find the desired solution is T_{ind}=0.26 seconds in total for individual models, compared to T_{MU}= 0.051 seconds for $M_{U.}$ The time to solve the maximization problem increases as the number of individual models per family (i.e., *INDV*) increases. For instance, for *INDV*=M, T_{ind} becomes 1.36 seconds, compared to T_{MU}= 0.104 seconds. For *INDV*=L, the total time it takes to find solutions for individual models, one model at a time is T_{ind}= 3.66 seconds, compared to T_{MU}=0.19 seconds. Finally, for *INDV*=XL, T_{ind}=8.20, whereas T_{MU} is equal to 0.73. Time speedups, with and without $T_{construct}$, are illustrated in Figure 72. As shown in the figure, even after considering the time $T_{construct}$ needed to build union models of different number of individual models (i.e., *INDV*), the time speedup is still a greater than 1. The time speedup obtained when neglecting $T_{construct}$ becomes more noticeable, with the minimum speedup of 5.10 (= 0.26/0.051) resulting from *INDV*=S, and the maximum speedup, achieved for *INDV*=L, equals 19.27 (=3.66/0.19).

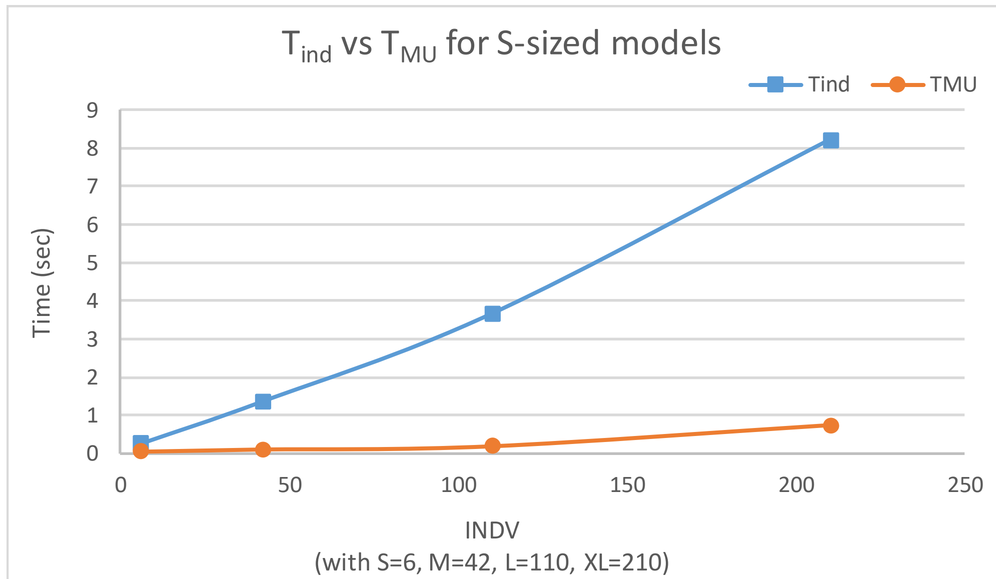


Figure 71 A comparison between T_{ind} and T_{MU} for small-sized models (S), with different numbers of models/families

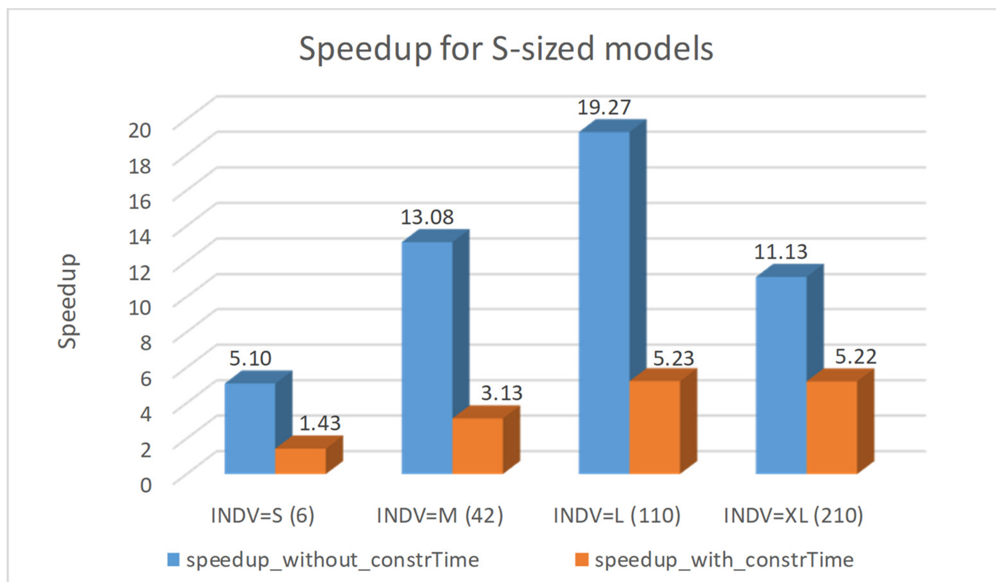


Figure 72 Speedup achieved by an M_U of small-sized model (S)

Figure 73 shows the time saving, in seconds, achieved by using M_U in comparison to using a set of individual models. From this figure, it can be noticed that the time saving increases with the increase of the number of models in a family. Although the time saving here is a few seconds, this could be important for real-time applications where every second counts, or when verification is offered as an online service, with many concurrent clients.

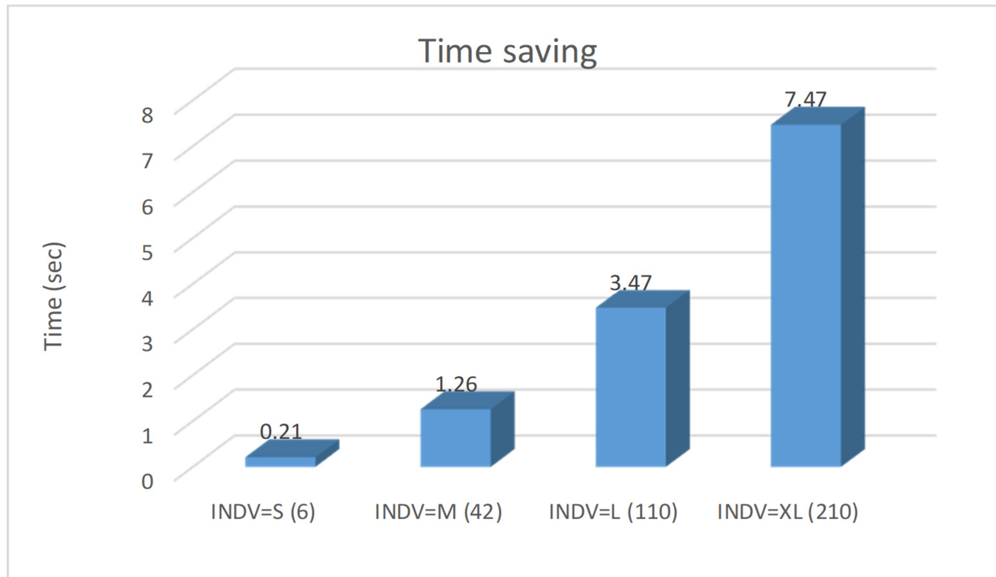


Figure 73 Time saving achieved by an M_U of small-sized model (S)

Finally, Table 6 and Figure 74 illustrate the percentage of memory consumption by M_U of small-sized models in relation to the memory used by the largest individual model in that family (i.e., *Max(MemoryIndv)*). For this experiment, the maximum memory used to store the largest individual model was 4.1 MB. On the other hand, the memory used by M_U increases with the number of individual models constituting M_U (i.e., *INDV*), as shown in Table 6.

Table 6 Memory consumption by an M_U of small-sized model

INDV	MemoryM$_U$	% of Memory consumption
S=6	4.7	(4.7–4.1)/4.1×100=14.6
M=42	5.9	(5.9–4.1)/4.1×100= 43.9
L=110	8	(8–4.1)/4.1×100= 95.1
XL=210	15.6	(15.6–4.1)/4.1×100= 280.5

Figure 74 visualizes the percentage of memory consumption by M_U of different *INDV* category. The maximum percentage is 280.5% for *INDV*=XL (i.e., 210 models in a family). This percentage indicates that although there are 210 models in the XL category, we only needed 3.8 (=15.6/4.1) times the memory of the largest individual model to process a model family of small-sized models and INDV=XL.

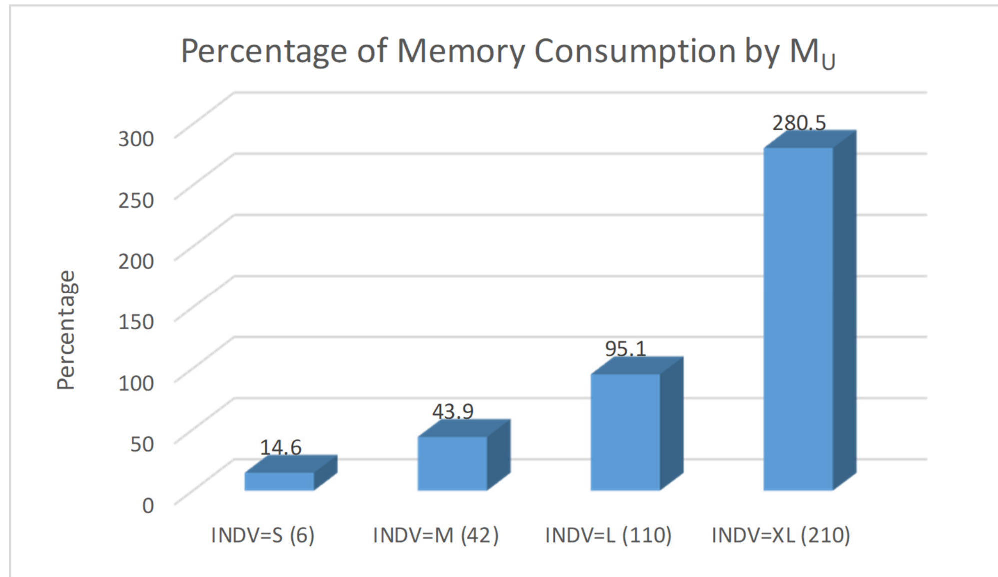


Figure 74 Additional memory consumed by an M_U of small-sized models (S), as a percentage of the memory consumed by the largest individual model

6.5.3 Results for SIZE=M and INDV=(S, M, L,XL)

Figure 75 illustrates the time needed to perform constrained maximization on a set of goal models of *SIZE*=M, where the number of elements per model ranges from 31 to 110 elements. The figure clearly shows the increase in T_{ind} as the number of models in a family increases, with the minimum T_{ind}= 0.29 sec obtained when *INDV*=S, and the maximum T_{ind}= 9.84 sec obtained with *INDV*=XL, which is 34 times higher than the minimum. T_{MU} also increases as the parameter *INDV* increases, but the increase of T_{MU} is not as noticable as in the case of T_{ind}. As can be seen from Figure 75, the minimum T_{MU} is 0.055 sec, and the maximum T_{MU} is 0.77 sec. These results suggest that the performance of analyzing individual models (in terms of the time needed to find the desired solution), one model at a time, degrades noticeably as the number of individual models increases. On the other hand, the use of M_U to analyze models, all at once, is scalable and time efficient, and does not worsen radically, even thought *INDV* increases from S to XL.

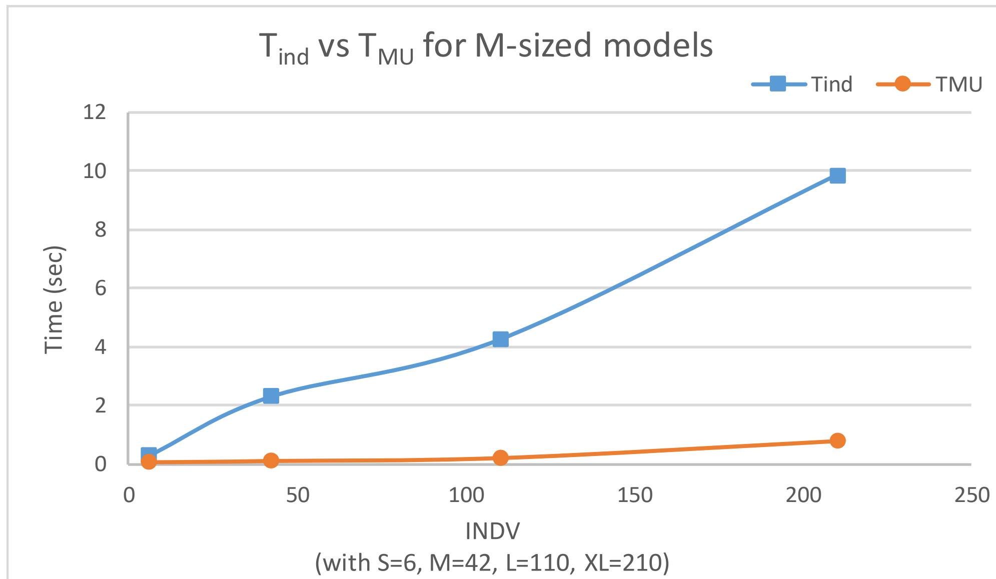


Figure 75 A comparison between T_{ind} and T_{MU} for medium-sized models (M), with different numbers of models/family

Figure 76, Figure 77, and Figure 78 respectively show the time speedup (with and without $T_{construct}$), the time saved and the memory consumed by using M_U of medium-sized models. As illustrated in Figure 76, analyzing a set of models using their M_U achieves noticeable time speedups as opposed to analyzing individual models separately. This speedup remains positive even after considering the time needed to build union models (i.e., $T_{construct}$).

Figure 77 shows the time saving achieved by using M_U compared to using a set of individual models, with the time savings becomes more evident as the number of models in a family increases.

Finally, Table 7 and Figure 78 illustrate the percentage of memory consumption by M_U of medium-sized models in relation to the memory used by the largest individual model in that family (i.e., *Max(MemoryIndv)*). For this experiment, the maximum memory used to store the largest individual model was 4.9 MB. On the other hand, the memory used by M_U increases with the increase of the number of individual models composing M_U (i.e., *INDV*), as shown in Table 7. Figure 78 illustrates the percentage of memory consumption by M_U of different *INDV* category. The maximum percentage is 214.3% for *INDV*=XL (i.e., 210 models). This percentage indicates that although there are 210 models in the XL category, we only needed 3.14 (=15.4/4.9) times the memory of the largest individual model to process a model family of medium-sized models and *INDV*=XL.

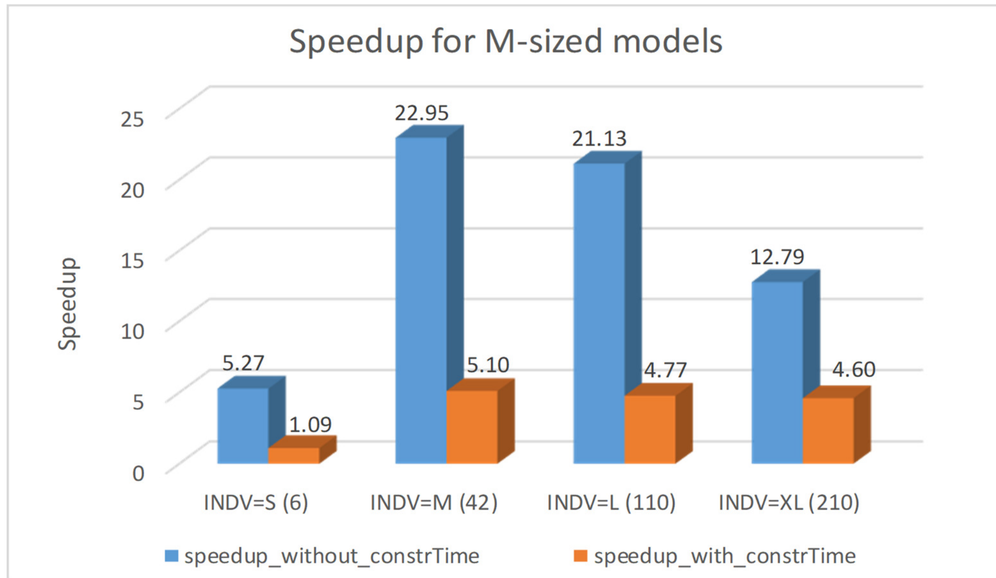


Figure 76 Speedup achieved by an M_U of medium-sized models (M)

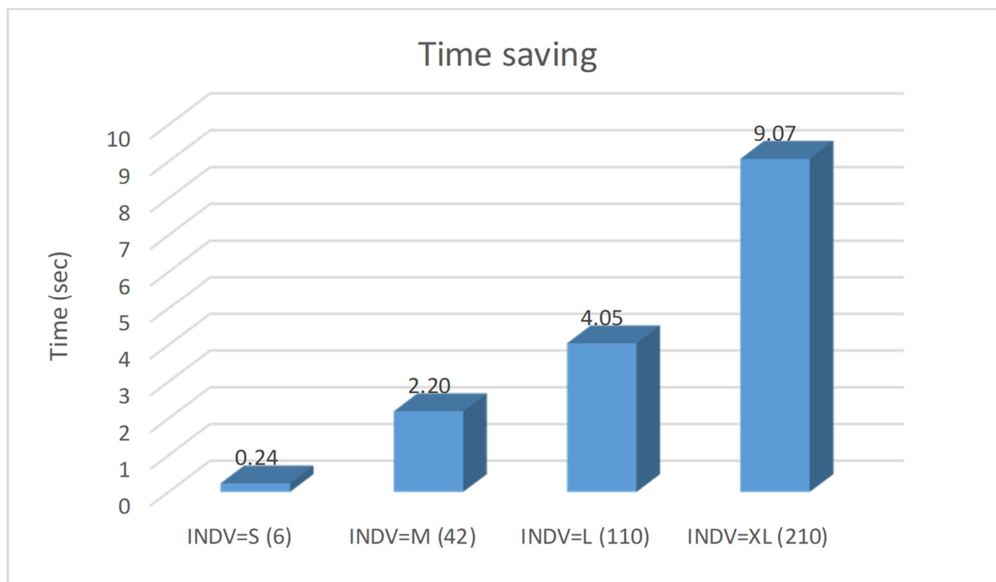


Figure 77 Time saving achieved by an M_U of medium-sized models (M)

Table 7 Memory consumption by an M_U of medium-sized model

INDV	MemoryM_U	% of Memory consumption
S=6	5.4	(5.4–4.9)/4.9×100=10.2
M=42	6.8	(6.8–4.9)/4.9×100= 38.8
L=110	10	(10–4.9)/4.9×100= 104.1
XL=210	15.4	(15.4–4.9)/4.9×100= 214.3

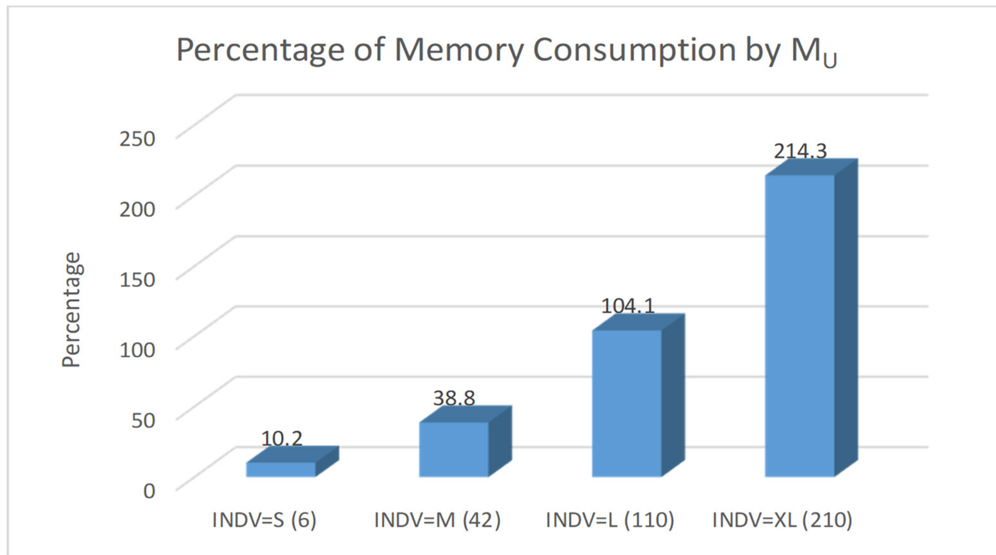


Figure 78 Additional memory consumed by an M_U of medium-sized models (M), as a percentage of the memory consumed by the largest individual model

6.5.4 Results for SIZE=L and INDV=(S, M, L,XL)

Figure 79 demonstrates results of the experiments conducted on models of *SIZE*=L, with different categories of *INDV* parameter. These results are compatible with the results obtained previously. In other words, the average T_{ind} and T_{MU} increase as the number of models in a family increases. For instance, T_{ind} increases from 5.34 sec for *INDV*=L to 10.86 sec for *INDV*=XL, while T_{MU} increases from 0.49 sec for *INDV*=L to 1.9 sec for *INDV*=XL.

The time speedup without considering $T_{construct}$, illustrated in Figure 80, also follows almost the same rhythm as for the previous two experiments, with speedup=5.08 for *INDV*=S, 15.46 for *INDV*=M, 10.92 for *INDV*=L, and finally 5.64 for *INDV*= XL. From the same figure, the *speedup_with_constrTime* is obviously lower than the one without $T_{construct}$, and it is positive (i.e., greater than 1) for all *INDV* categories except for *INDV*=S, where the speedup is 0.63. This value indicates that the time T_{MU} plus $T_{construct}$ exceeds that time T_{ind}.

Figure 81 shows the time saving achieved by using M_U. The figure illustrate clearly that the more the number of models in a family, the more the savings in terms of runtime.

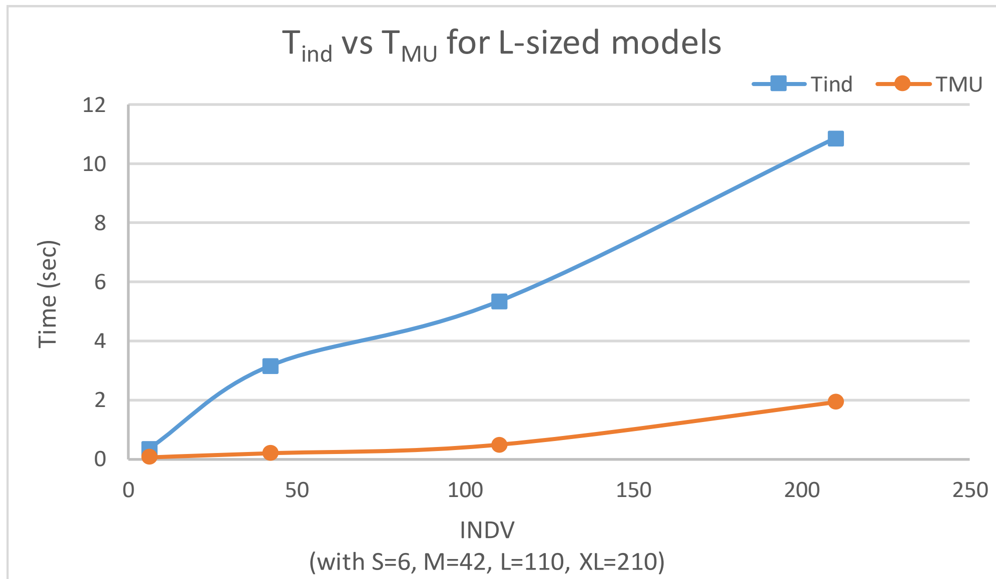


Figure 79 A comparison between T_{ind} and T_{MU} for large-sized models (L), with different numbers of models/family

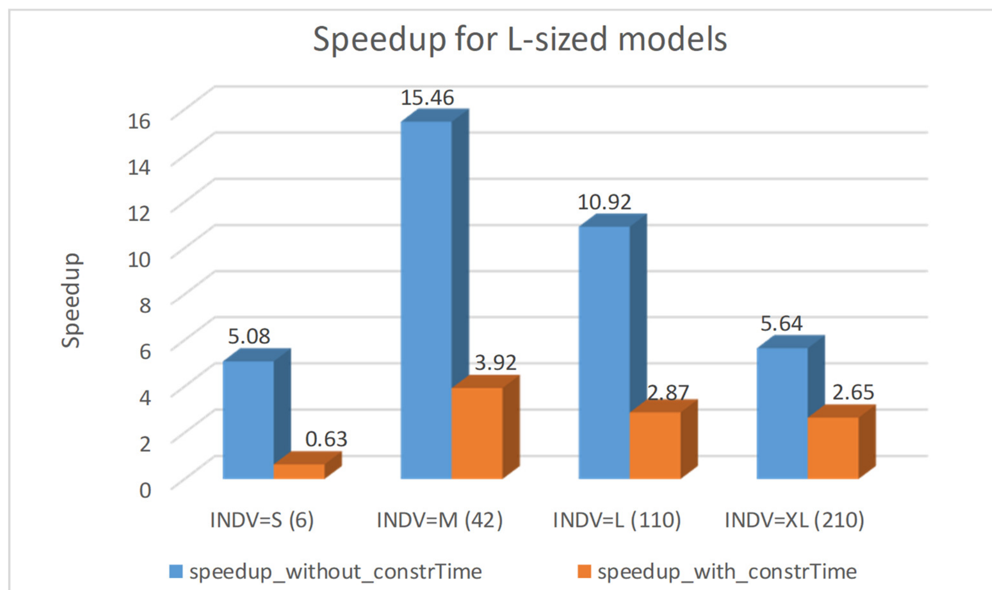


Figure 80 Speedup achieved by an M_U of large-sized models (L)

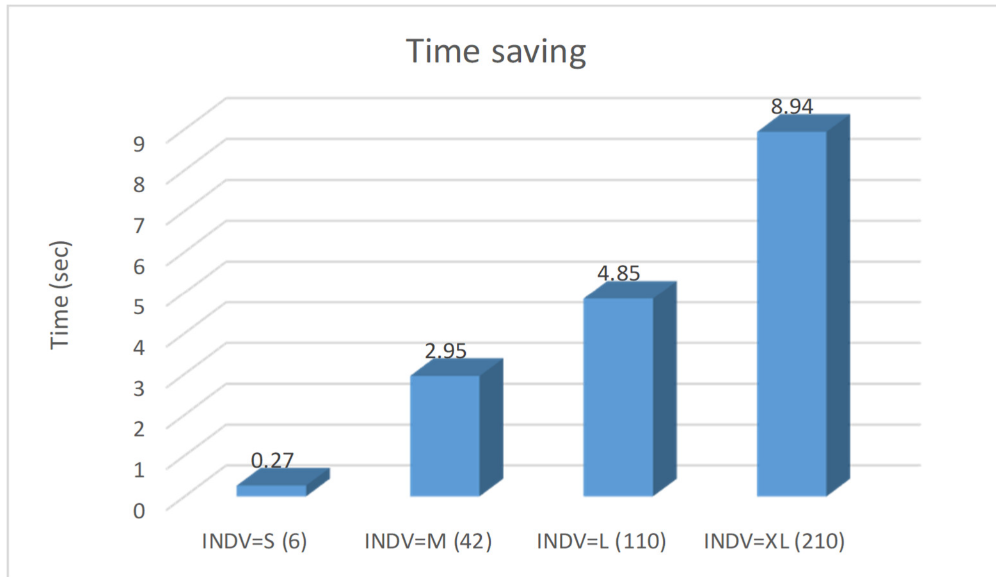


Figure 81 Time saving achieved by an M_U of large-sized models (L)

Figure 82 illustrate the percentage of memory consumption by M_U of large-sized models compared to the memory used by the largest individual model in that family. For large-sized models, the maximum memory used to store the largest individual model was 6.7 MB. Table 8 summarizes the memory used by M_U (MemoryM_U) and shows that the latter increases with the increase of the *INDV*. From Figure 82, the maximum percentage is 222% for *INDV*=XL. This percentage suggests that the union model of the largest family needed only 3.22 (=21.6/6.7 MB) times the maximum memory needed to store and process the largest individual model.

Table 8 Memory consumption by an M_U of Large-sized model

INDV	MemoryM_U	% of Memory consumption
S=6	7.1	(7.1–6.7)/ 6.7×100=5.97
M=42	9.4	(9.4–6.7)/ 6.7 ×100= 40.30
L=110	14.8	(14.8–6.7)/ 6.7×100= 120.90
XL=210	21.6	(21.6–6.7)/ 6.7×100= 222.39

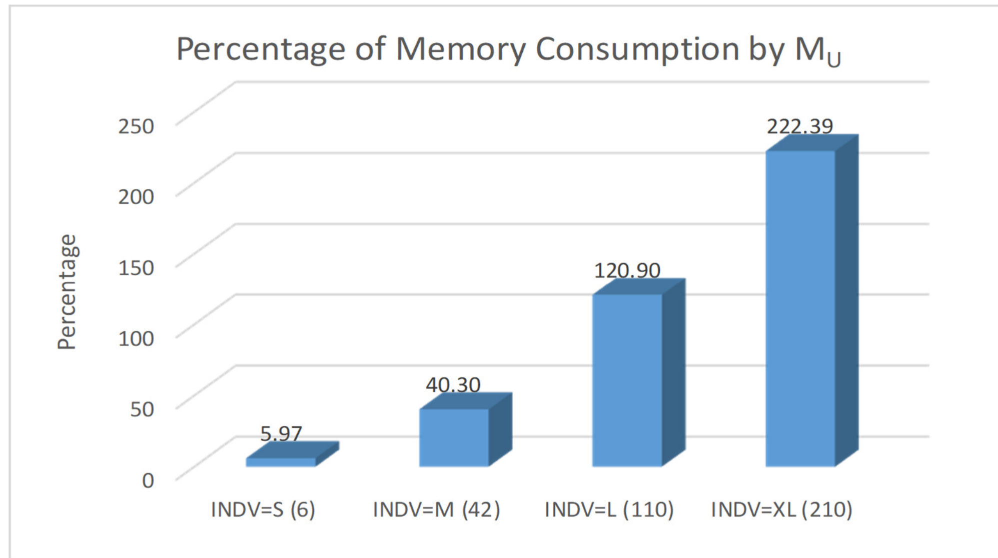


Figure 82 Additional memory consumed by an M_U of large-sized models (L), as a percentage of the memory consumed by the largest individual model

6.5.5 Results for SIZE=XL and INDV=(S, M, L, XL)

This section reports on the results of performing constrained maximization on very large models of *SIZE*=XL, where the number of elements per model ranges from 421 to 1640 elements. The results illustrated in Figure 83 show that T_{ind} increases with the increase of the number of models in a family. On the other hand, T_{MU} scales well and increases slightly under the same settings. Figure 84 demonstrates the speedup (with and without $T_{construct}$) achieved by using an M_U that captures models of *SIZE*=XL, along with different categories of parameter *INDV*. Without considering $T_{construct}$, the maximum speedup=19.5 is obtained when *INDV*=L while the minimum speedup is 7.3 with *INDV*=S. When considering $T_{construct}$ in the calculation of speedup, the results become lower, and show a behavior close to the previous experiment, where all speedups where greater than 1, except when *INDV*=S. Here again, the less-than-one speedup value means that the use of M_U, when taking its construction time into account, does not achieve any speedup.

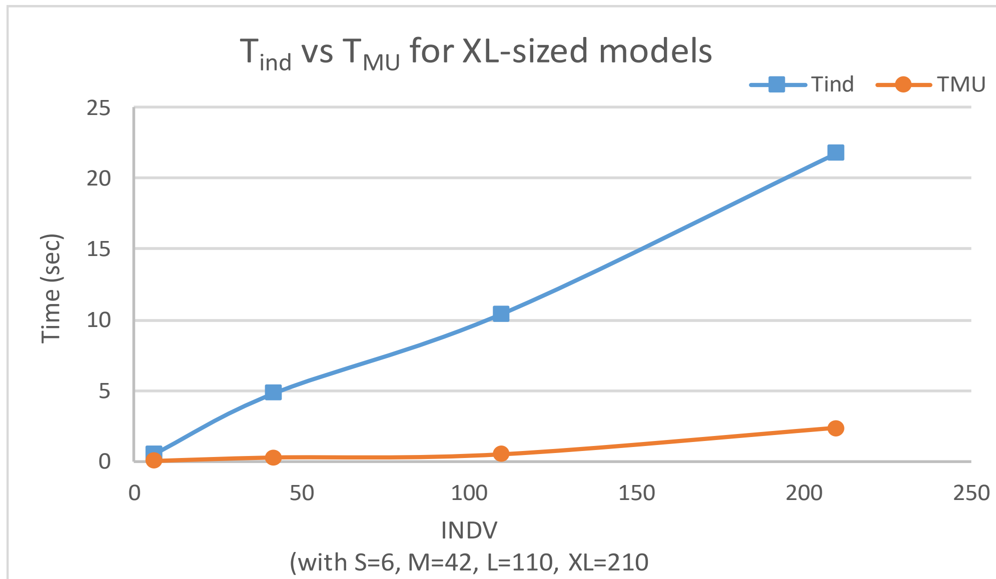


Figure 83 A comparison between T_{ind} and TM_U for XL-sized models, with different numbers of models/family

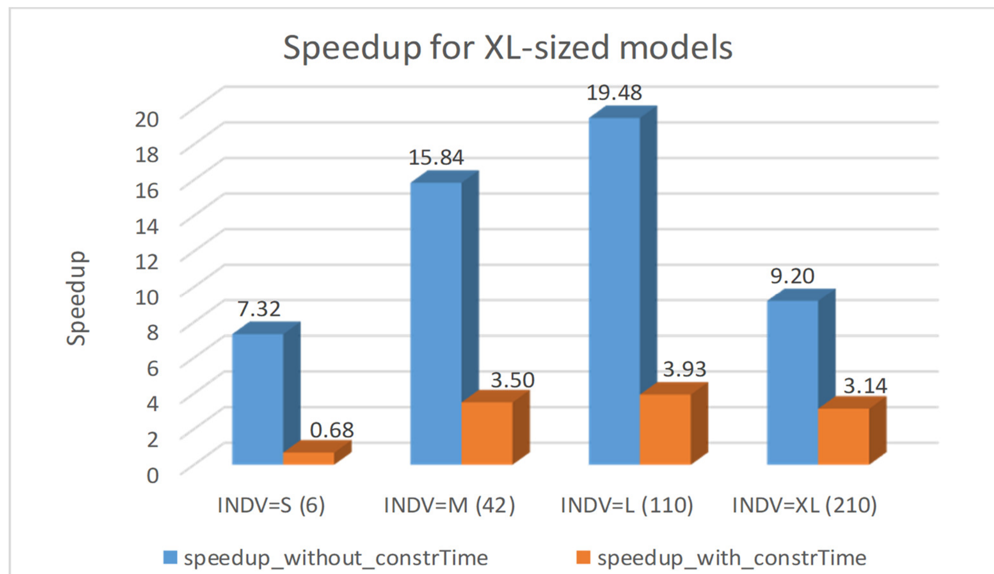


Figure 84 Speedup achieved by an M_U of XL-sized models

Time savings achieved by using M_U are depicted in Figure 85. The figure illustrates clearly that the higher the number of models in a family, the higher the savings in terms of runtime, and this consistent with all of the previous experiments. Finally, Figure 86 illustrates the percentage of memory consumption by M_U of XL-sized models in relation to the memory used by the largest individual model in that family. For XL-sized models, the maximum memory used to store the largest individual model is 7.9 MB. Table 9 summarizes the

memory used by M_U and Figure 86 shows that this memory consumption increases as *INDV* gets larger. From Figure 86, the maximum percentage is 273% for *INDV*=XL. This percentage suggests that the union model of the largest family needed only 3.7 (= 29.5/7.9) times the maximum memory needed to store and process the largest individual model of *INDV*=XL and *SIZE*=XL.

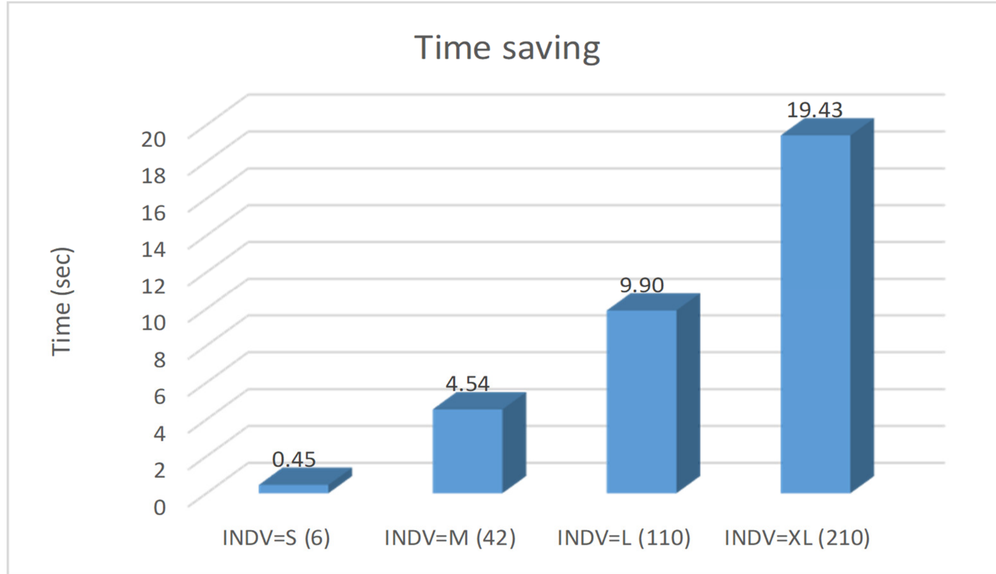


Figure 85 Time saving achieved by an M_U of XL-sized models

Table 9 Memory consumption by an M_U of XL-sized model

INDV	MemoryM_U	% of Memory consumption
S=6	8.4	(8.4–7.9)/ 7.9×100=6.33
M=42	10.2	(10.2–7.9)/ 7.9×100= 29.11
L=110	15.3	(15.3–7.9)/ 7.9×100= 93.67
XL=210	29.5	(29.5–7.9)/ 7.9×100= 273.42

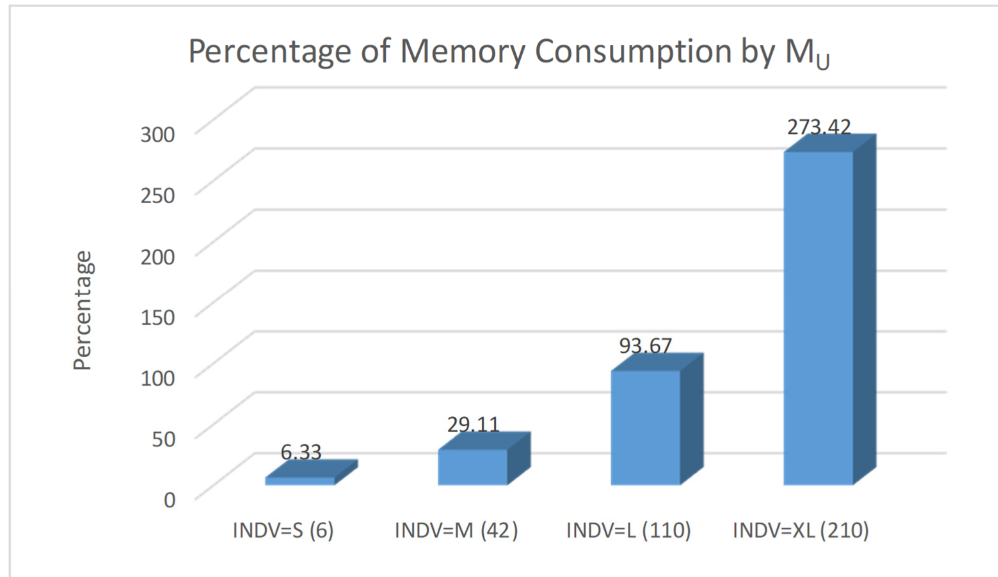
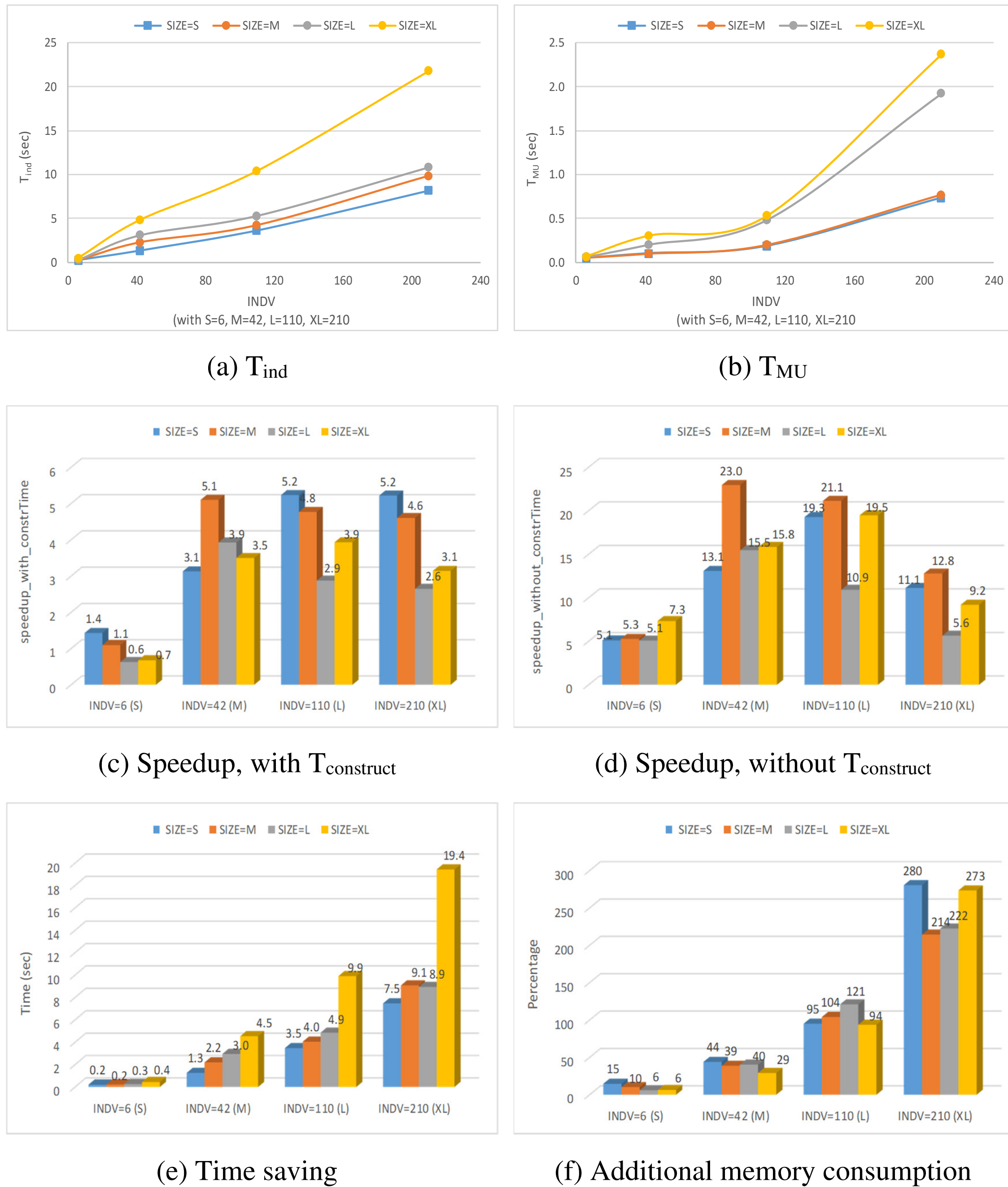


Figure 86 Additional memory consumed by an M_U of extra-large models (XL), as a percentage of the memory consumed by the largest individual model

6.5.6 Summary of All Results

This section compares all results together in order to obtain observations about the behavior of T_{ind}, T_{MU}, as well as speedup achieved from using M_U. Figure 87 (a) summarizes T_{ind} observations needed to analyze individual models of all categories of parameters *SIZE* and *INDV*. It can be noticed that T_{ind} grows as both *SIZE* and *INDV* increase. Figure 87 (b) shows observations for T_{MU}, where the latter increases in a linear manner, without an increase as radical as for the case of T_{ind}. From these results, we conclude that using a union model to analyze groups of models, all at once, is scalable and efficient, especially with large number of models, that have also large number of elements.

The summaries of time speedups and time savings illustrated in Figure 87(c)-(e) also support our argument regarding the feasibility and benefits of using M_U, for any family size. Finally, Figure 87 (f) indicates that the use of M_U comes with at cost, which is memory consumption, but this cost is tolerable as it does not proliferate radically compared to the increase of the number of individual models in a family.



(a) T_{ind}

(b) T_{MU}

(c) Speedup, with $T_{construct}$

(d) Speedup, without $T_{construct}$

(e) Time saving

(f) Additional memory consumption

Figure 87 A summary of T_{ind} for all SIZE categories

6.6. Chapter Summary

This chapter investigated RQ3 by evaluating, empirically, the feasibility of using union models to perform backward propagation on GRL model families, implemented as

constrained optimization on a commercial tool (IBM CPLEX). The chapter first recalled the original semantics of both GRL forward and backward propagation algorithms. It then contributes many important tools and results:

- An adaptation of the GRL arithmetic semantics to support forward and backward propagation of model families.
- The implementation of the algorithm to create a union model from partial GRL models, with an export mechanism to CPLEX OPL for individual and union models.
- An experiment (targeting 4 family sizes) that demonstrate the usefulness of union models in analyzing GRL model families, all at once, compared to individual models.
- The chapter also reported on the speedups and time savings gained from using M_U, which generally increase as models become larger and as model families include more models.

The next chapter goes back to an important problem identified earlier in the thesis, namely that a union model may not be a valid instance of the metamodel of the individual models. It also proposes two methods to determine and predict non-compliant locations and minimally relax the metamodel accordingly.

Chapter 7. Metamodel Relaxation to Support Model Families

In the previous chapters, we have proposed *union models* that capture the union of all models in a model family in order to enable a more efficient analysis of family members, all at once, where the analysis was done at the graph level. However, as discussed in Section 1.3, despite having each model in a model family conforming to the same metamodel, there is still *no* guarantee that their union model will conform to the original metamodel of the family members (as illustrated already in Figure 6). This is because a union model could contain elements, such as links or attributes, that violate multiplicity constraints or other external OCL constraints of the original metamodel.

This chapter aims to enable analysis closer to the language itself and also to support the representation of union models M_U as valid instances of a metamodel, by inferring a *minimally-relaxed* metamodel MM_U (to which a union model conforms) from the structure of the original metamodel MM (to which all family members conform). In particular, instead of having a brute force relaxation of all metamodel constraints, we contribute a heuristic method that relaxes *specific* constraints in MM (related only to multiplicities of attributes and association ends), by *inferring* where such relaxations (referred to as *relaxation points*) are needed in the metamodel. To infer relaxation points, structural patterns are first identified in metamodels, then an evidence-based or an anticipation-based approach is applied to get the actual inference. The objective of inferring particular relaxation points in MM is to be able to adapt the modeling tools and analysis techniques that exist for a given modeling language *once* and *minimally* for all potential model families of that language, and hence to minimize adaptation effort.

Different approaches were proposed in the literature to relax constraints imposed by a language's metamodel to facilitate/perform a particular task. These approaches are already discussed in Section 2.13. The rest of the chapter is organized as follows: Section 7.1 provides different scenarios, from two languages, that justify the need to relax metamodels. Sections 7.2 and 7.3 discuss the evidence-based and anticipation-based solutions,

respectively, proposed for metamodel relaxation. Finally, Section 7.4 summarizes the chapter.

The contents of this chapter have been introduced in two papers [185][186] and expanded in a third one [225]. Please note that the formalization of models and union models introduced in Chapter 4 can be used in this chapter, but the results presented here do not depend on this formalization and can be adapted to more conventional descriptions such as EMF and MOF.

7.1. Why Metamodel Relaxation?

This section provides three illustrative examples from two modeling languages: GRL and UML class diagrams. These examples motivate the need to relax some of the metamodel multiplicities and other constraints to support model families (RQ4).

7.1.1 Scenario One: Different GRL Actors with the Same Goal

In this section, an evolution scenario for simple GRL models is given. As shown in Figure 88(a), the first version of the GRL model represents an actor (ActorA) containing an intentional element of type goal (SomeGoal).

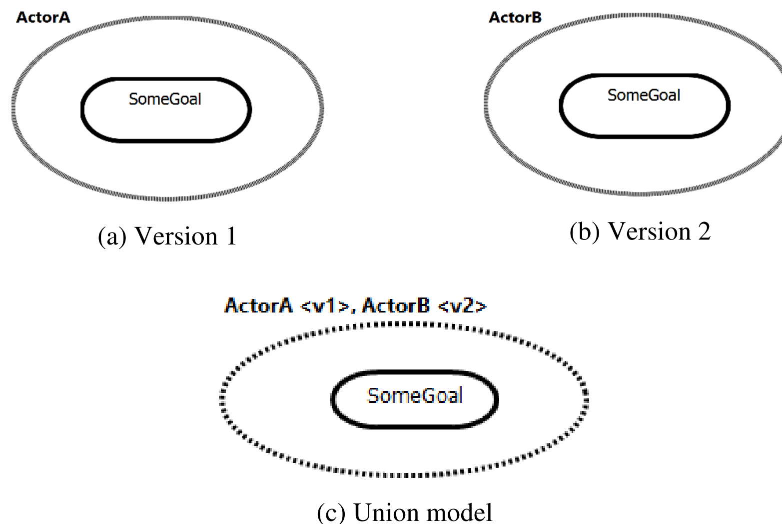


Figure 88 First GRL example: SomeGoal moves from ActorA to ActorB with their union model and relevant GRL metamodel elements

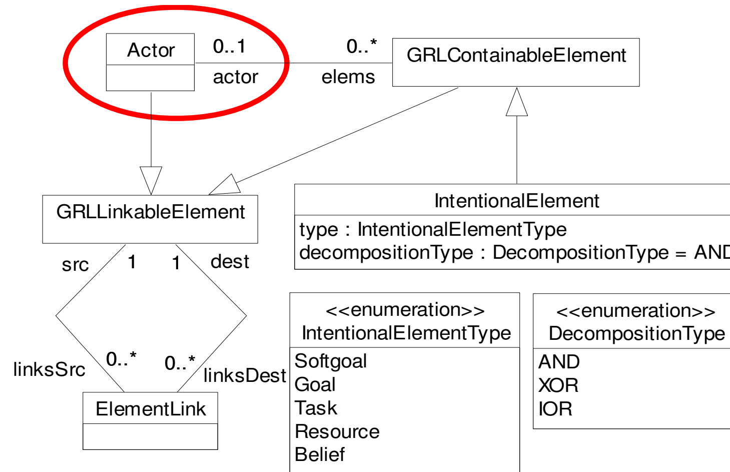


Figure 89 Extract of the GRL metamodel

Let us assume that this model evolves over time, such that SomeGoal is moved from ActorA to ActorB. The evolved version is shown in Figure 88(b). Each of the model versions, *separately*, conforms to the metamodel excerpt represented in Figure 89, since an intentional element may be included by 0 or 1 actor.

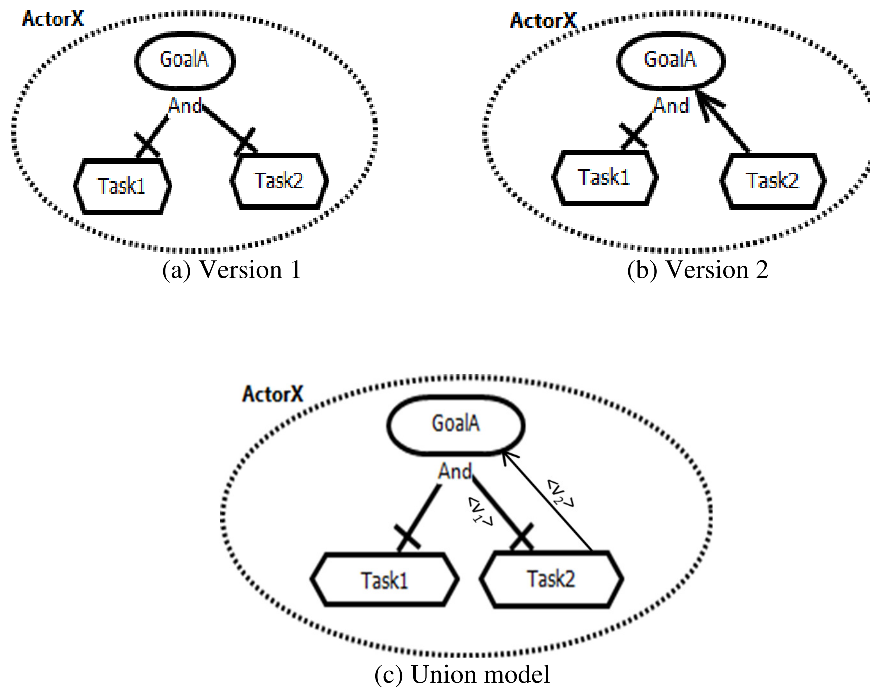
However, if we want to aggregate both models into one union model to represent their model family, as illustrated in Figure 88(c), the resulting union model will not conform to the current metamodel. This is because an intentional element (SomeGoal) cannot be contained by more than one actor. This violates the multiplicity constraint circled in the metamodel excerpt (Figure 89).

One way to re-establish conformance is to relax the multiplicity constraint of the actor association end to be (0..*) instead of (0..1). The purpose of this relaxation is to allow different actors in different model versions to contain the same intentional elements. Such new relaxed MM_U metamodel would hence allow the union model (M_U) and the individual family members (versions 1 and 2) to be conform.

7.1.2 Scenario Two: GRL Intentional Elements with Multiple Links

The example in Figure 90 shows another evolution scenario in GRL. In the first version (Figure 90(a)), an intentional element of type goal (GoalA) is decomposed into two intentional elements of type task (Task1 and Task2). Let us assume this model evolves such that the type of link that associates GoalA with Task2 changes from a decomposition link to a

contribution link (as shown in Figure 90(b)). Decompositions and contributions are two different sub-types of *ElementLink* in the metamodel of Figure 89.



(a) Version 1 (b) Version 2

(c) Union model

Figure 90 (a) Version 1: GoalA and Task2 with decomposition link (b) Version 2: GoalA and Task2 with Contribution Link (c) union model that regroups both models

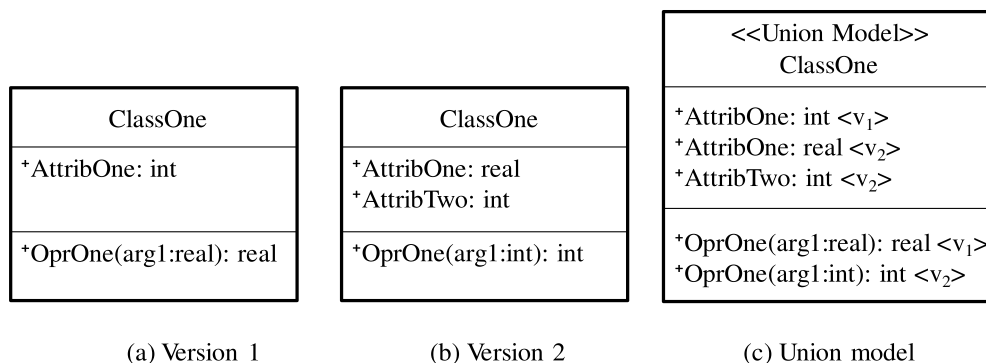
The union model that captures both models is shown in Figure 90(c). However, this union model cannot be represented by the original metamodel because of an additional (OCL) constraint stating that any pair of *IntentionalElements* can be connected by at most one *ElementLink* (Figure 89). Hence, it is not allowed to have both a decomposition link and a contribution link connecting GoalA with Task2. Therefore, to support the representation of such union models, the OCL constraint of the original metamodel *MM* needs to be relaxed (or removed) in the evolved metamodel MM_U to allow more than one element link to connect the same pair of intentional elements in the aggregated M_U family model.

It is worth mentioning that in GRL, annotations are directly supported with "metadata" attached to any model element. However, should an annotation mechanism not be available in the source modeling language, the relaxed MM_U metamodel may also need to be extended to include such a mechanism (e.g., STAL annotations).

7.1.3 Scenario Three: UML Attributes with Multiple Types

This third evolution scenario focuses on UML class diagrams [161]. Let us assume that the first version (Figure 91(a)) of class ClassOne has an attribute AttribOne of type int, as well as an operation OprOne, with argument arg1 of type real, that returns a parameter of type real. This class evolves to version 2 (Figure 91(b)) such that the type of AttribOne becomes real instead of int, and the operation's parameter and return data types become int (instead of real). These two versions are aggregated into a union model, illustrated in Figure 91(c), where version-based annotations are also used on attributes and operations.

Having multiple operations with the same names but different argument types is allowed in UML. However, UML enforces stronger constraints on attribute names. The resulting union model violates the UML standard metamodel since the latter does not support the representation of one attribute with multiple data types. In order to re-establish conformance, the multiplicity constraint related to attributes could be relaxed in UML such that attributes would be allowed to have a *collection* of types instead of only one type.



(a) Version 1 (b) Version 2 (c) Union model

Figure 91 Two UML classes and their resulting union model

7.2. Proposed Solution 1: Evidence-Based Relaxations

This section and the following one discuss our proposed solutions for the metamodel relaxation problem. The first solution is the **evi**dence-based **me**tamodel **re**laxation to support model **fam**ilies (*Evi-MeReFam).* Evi-MeReFam is a three-phase approach (Figure 92) for metamodel relaxation based on evidence provided by the usage pattern of a language and the evolution/change behavior of *current* model families of that language. In this chapter,

we use the GRL as the primary modeling language to illustrate the main concepts of our proposed solutions.

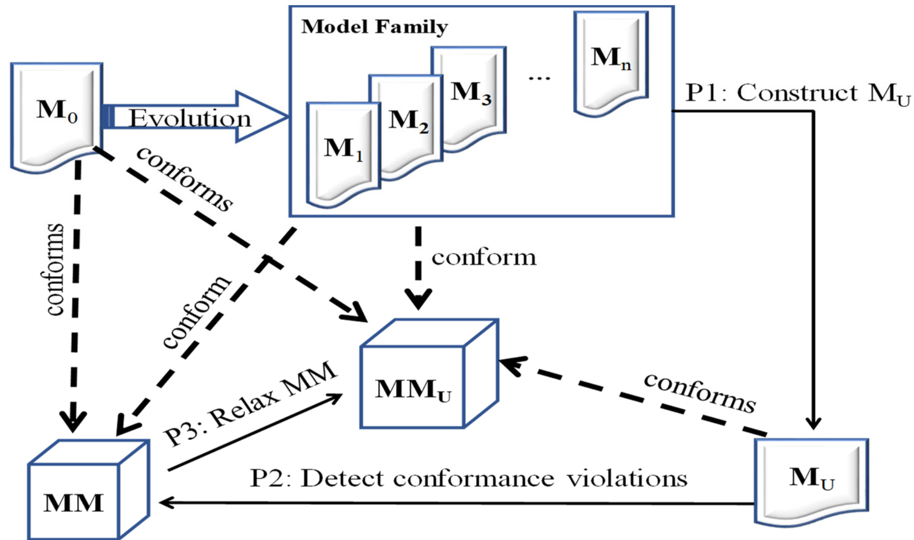


Figure 92 A Conceptual Architecture of the Evi-MeReFam Method.

7.2.1 Phase One: Construct the Union Model

The objective of this phase is to aggregate the various models of a model family into one single union model, M_U. Let M_0 be the initial version of a model that conforms to *MM* and evolves over time or across product variants into several versions, M_1, M_2, M_3, etc. Then a model family, *MF,* is $MF = \{M_0, M_1, M_2 ... M_n\}$, where n is the number of member models in a model family. As discussed in Section 4.4.2, the union model M_U of this MF is: $M_U = \bigcup_{i=0}^{n} e_i \in M, \forall\, Mi \in MF$.

Figure 93 illustrates the M_U of three models, where elements of M_U are annotated with information about model versions. Although a concrete annotation syntax is used here for illustration purpose, annotations are in fact added to the object model.

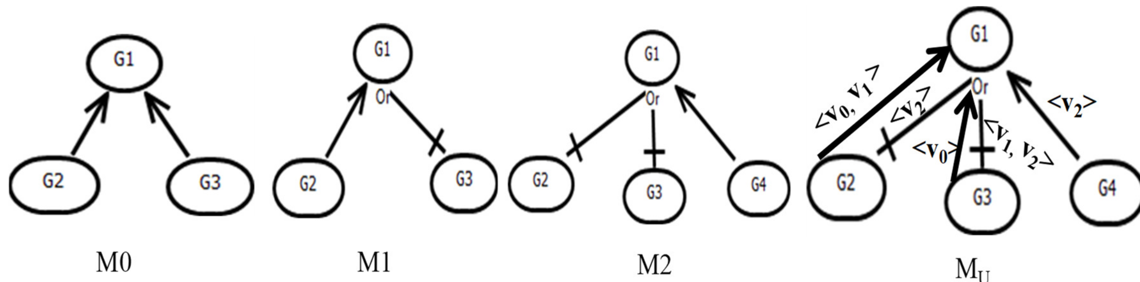


Figure 93 The set of models in MF (M_0, M_1, and M_2) and their union model M_U.

While constructing a union model, we observe the changes that happen across models in a model family in order to decide how to relax a metamodel accordingly. For all models in a model family (of any language), we can formalize changes (i.e., deltas) between models to be the set D, such that $D \rightarrow M \times M$, for any model M in M_U.

To better understand the nature of changes that happen in a model family, one intuitive option is to represent D in operational terms, i.e., as a sequence of transformations that add, remove, or modify elements (i.e., nodes and edges) of models. Hence, a delta $\Delta \in D$ can be viewed as a sequence of transformations that, when needed, can be applied to a model M_i to yield a model M_{i+1}, as follows: $M_i + \Delta = M_{i+1}$.

To characterize changes in model families of any language, and inspired by the approach proposed by Alanen and Porres [62], we have identified six elementary transformations that will be used as the basis for defining D. We assume that it is not possible to change the type of a model element, e.g., an instance of a UML class cannot become an instance of a package, and an element cannot change its unique identifier (ID). The operations in D are:

- Node/edge creation and deletion:
    - *add(N/E, t)*: add a new node *N* or edge *E* of type *t*, with a unique identifier ID.
    - *del(N/E, t)*: delete a node *N* or an edge *E* of type *t*, with a unique identifier ID.
- Modification of an attribute of primitive type of nodes/edges:
    - *update (N/E, t, val1, val2):* set the value of node *N* of type *t* from value *val1* to *val2*.

Such characterization of deltas is important since they are used in the next phase for matching against the categories of metamodel constraints to help infer metamodel relaxation points.

7.2.2 Phase Two: Detect Non-conformance

The purpose of this phase is to detect *current* conformance violations in the original metamodel, MM, caused by M_U. This is done by (1) characterizing the original metamodel

constraints, and (2) comparing deltas (obtained in phase 1) against metamodel constraints to check if any particular delta causes conformance violations with MM. More details are provided in Section 7.2.4.

Conformance between the original metamodel MM and the union model M_U is verified by checking if the co-existence of deltas of a particular category (obtained in phase 1) with other components in the same union model could cause a violation of a metamodel constraint. Conformance violations, in general, could stem from having two links between the same pair of elements (for instance, two different links between the same pair of GRL intentional elements). Another source of violations is when the same attribute has two different data types, and both data types need to be considered in the M_U. In this scenario, multiplicity constraints of attributes' data types are violated. If non-conformance is detected, phase 3 takes place.

7.2.3 Phase Three: Infer Metamodel Relaxation Points

Based on the metamodel conformance violations (detected in phase 2), the modeler is now able to decide on two things: relaxation types and locations of the relaxation points. Regarding the *relaxation types*, we observe that the evolution of models in a family does not involve adding new concepts. Rather, it involves changes in the number (i.e., multiplicity) of links and/or attributes. Hence, a modeler needs only to relax the metamodel's internal constraints that are related to multiplicities of attributes and association ends and/or some of the external (OCL) well-formedness constraints.

Regarding the locations of *relaxation points*, only those constraints that are directly affected by violations (detected in phase 2) will be relaxed. For instance, if a conformance violation is caused by the existence of two links between the same pair of elements, then the multiplicity constraint of association ends between these elements needs to be relaxed in the metamodel. To ensure that we do not relax unnecessary constraints, and hence have as few relaxations as possible, only the *upper bound* of constraints that are directly violated by deltas are relaxed to their maximum bound. For example, the multiplicity constraint 2..5 is relaxed directly into 2..* so as to permit any future addition of the same type of delta in advance. In other words, even thought it could be required (based on the currently available models) to only relax the current 2..5 constraint into 2..6; this constraint may soon need to

be altered again into 2..7, 2..8, 2..9, etc. due to considering more models. Hence, to avoid multiple relaxations (which are indeed costly in terms of adapting tools each time there is a relaxation), we decide to relax the constraint's upper bound right away into the * value. In this context, lower bounds of the constraints do not need to be relaxed. By this, we achieve part of the "*minimal relaxation*" we target in this work.

Relaxing only the violated constraints means that some of the other constraints will not be altered (e.g., multiplicity constraints of association ends between container classes and their contained elements). This is decided based on the usage patterns of the modeling language and the way models in a model family change.

7.2.4 Illustrative Example of Evi-MeReFam

To illustrate Evi-MeReFam, we consider goal model families and the GRL metamodel (part of URN's metamodel [6]).

To detect non-conformance caused by GRL model families, we need to: (1) characterize the types of deltas encountered in a GRL model family, (2) characterize the constraints specified in a GRL metamodel, and (3) compare and match each type of delta with meta-model constraints to build an evidence about which constraints would be violated, and hence, need to be relaxed.

Evidence-based deltas encountered in a GRL model family (from each M_i to M_{i+1}) include the following:

- *D1*: Add a new ElementLink (e.g., contribution link or decomposition link).
- *D2*: Delete an existing ElementLink.
- *D3*: Modify an ElementLink's attribute by changing its value (e.g., change a quantitativeContribution value from 100 to 70 for the same contribution link, or a decomposition type from AND to OR).
- *D4*: Add a new IntentionalElement (goal, task, etc.).
- *D5*: Delete an existing IntentionalElement.
- *D6*: Modify an IntentionalElement by changing its name or attribute (e.g., modify the name of a goal from G1 to G2, or change an importanceQuantitative from 5 to 25).

- *D7*: Add a new Actor.
- *D8*: Delete an Actor.
- *D9*: Modify an Actor name (e.g., from Actor1 to Actor2) while keeping the same IntentionalElements that are contained by actors.

On the other hand, the GRL metamodel constraints are categorized into: (1) multiplicity constraints of association ends between two classes (elliptical constraints C1-C5 in Figure 94), and (2) multiplicity constraints on the allowed number of attributes' datatypes (rectangle constraints C6-C8 in Figure 94).

- C1: between Actors and GRLContainableElements.
- C2: between Actors and GRLspec.
- C3: between GRLContainableElements and GRLspec.
- C4: between ElementLink and GRLspec.
- C5: between ElementLink and GRLLinkableElements.
- C6: constraints on attributes contribution and quantitativeContribution, which can have only one value of types ContibutionType and Integer, respectively.
- C7: constraints on the allowed number of values for attributes type and DecompositionType.
- C8 constraints on the allowed number of values for attributes importance and importanceQuantitative.
- C9: Any external OCL constraint (to be characterized too).

After characterizing changes (D_i) in a GRL model family, and the constraints (C_i) in a GRL metamodel, we now compare and match each delta against each constraint. If any violation happens, this means that the existence of such delta in a model family causes a metamodel non-conformance issue, hence, phase 3 of the Evi-MeReFam takes place.

Based on our analysis of each delta, $D_i \in$ D, and its impact on constraints, C_i, we observe the following characterization of conformance/non-conformance issues in the GRL example:

- Any delta D_i that involves adding a new IntentionalElement (D4) or Actor (D7) will not violate MM constraints.
- Any delta D_i that involves deleting an element (i.e., D2 for ElementLinks, D5 for IntentionalElements, and D8 for Actor) does not have any impact on any constraint.
- Any delta D_i that involves adding a new ElementLink between the *same pair* of IntentionalElements will violate external OCL constraints (C9).
- Some modification deltas have a direct impact on internal MM constraints (or its explicit OCL constraints), regardless of the way of representing these deltas in M_U. These deltas include D6 and D9. For example, in D9, changing the Actor name from Actor1 to Actor2 while keeping the same goals that are contained by these actors would cause violations to C1. In this case, the upper bound of the association end between classes Actor and GRLContainableElement (from the actor side) would be relaxed from 0..1 to 0..*.

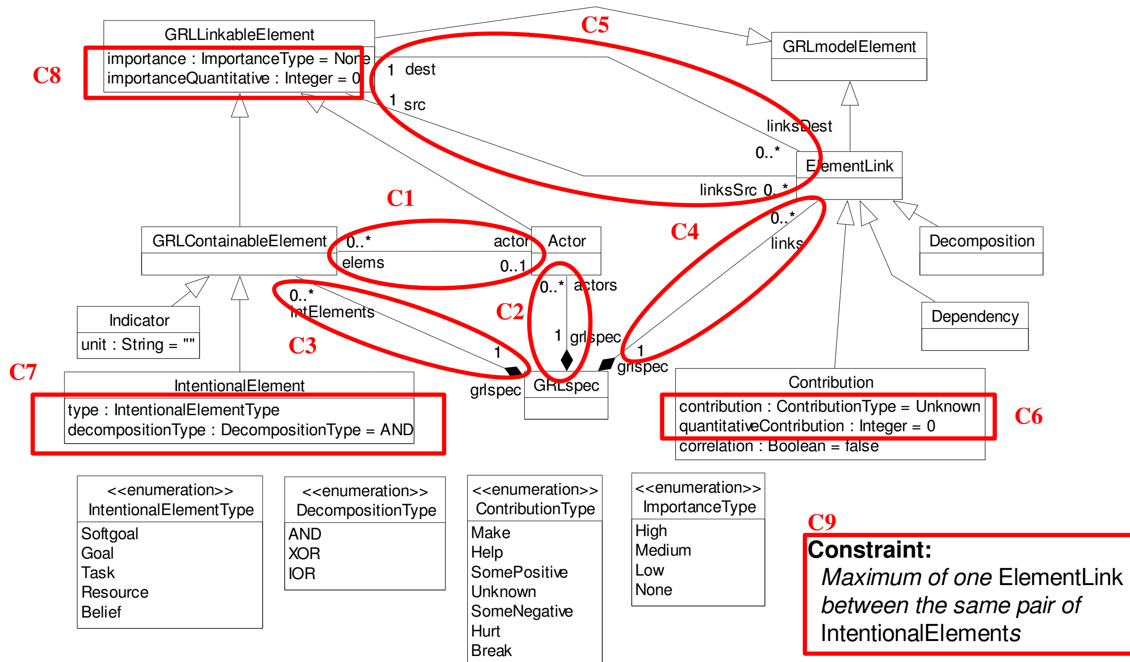
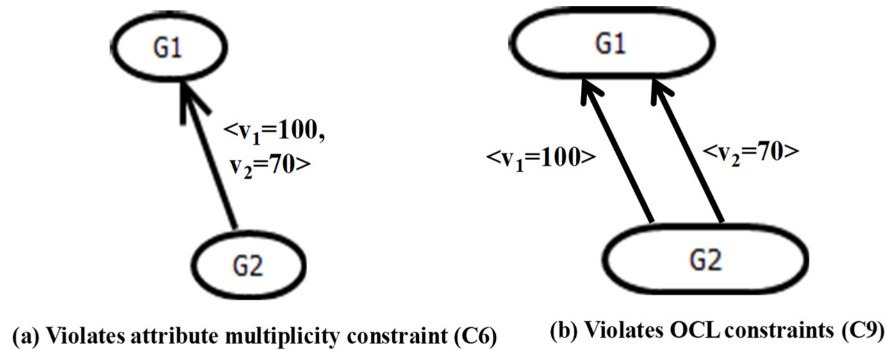


Figure 94 Characterization of GRL's metamodel (internal) constraints.

- Other modification deltas, i.e., D3 and D6, either affect the metamodel constraints or the language's external OCL constraints depending on the way of

merging model variations in the union model (as illustrated in Figure 95). For example, in D3, if we represent a change in a quantitativeContribution value (for the same contribution link) from 100 to 70 in M_U using one contribution link with two values, this will cause a conformance violation for constraint C6. Accordingly, the number of allowed datatypes for the attribute quantitative-Contribution needs to be relaxed from 1 Integer (Integer=0) to a set of Integers ({Integer}={0}). However, if we represent the change as two different contribution links, each link with a quantitative value and a version annotation, this will violate C9.



(a) Violates attribute multiplicity constraint (C6) **(b) Violates OCL constraints (C9)**

Figure 95 Examples where merging elements differently affects MM violations.

7.2.5 Limitations of Evidence-based Relaxation

In Evi-MeReFam, performing metamodel relaxation based only on matching deltas against constraints according to the delta/constraint characterization is an evidence-based solution with *local minimality*. In fact, a pure evidence-based solution provides insights about relaxation points related directly to the type of delta (detected in phase 2 of Evi-MeReFam) for a particular modeling language (GRL in our example) and for specific models in a family. Hence, if a new model is added to a model family, phases 2 and 3 need to be repeated to detect new changes and infer new relaxation points in the metamodel. This local minimality does *not* guarantee that relaxation is done only once, for all possible models in that language.

To overcome this issue and to infer other locations (i.e., relaxation points) of MM multiplicity constraints that actually need to be relaxed, *independently* of the models in a family, a naïve brute-force approach (that relaxes all multiplicity constraints and external

constraints in the metamodel) cannot be used either, as it would imply more extensive changes to existing tools and analysis methods than needed. Instead, relaxation points should be *anticipated* based on the structure and usage patterns of the language.

7.3. Solution 2: Anticipation-Based Relaxation

This section provides our second suggested solution for the metamodel relaxation problem, named **anti**cipation-based **me**tamodel **re**laxation to support model **fam**ilies (*Anti-MeReFam*). As its name indicates, the core of this approach is to *anticipate* relaxation points in a metamodel. The anticipation (or prediction) process conducted in this approach involves analysing the *structure* of the metamodel at hand and identifying similar *structural patterns*, either inside the same metamodel or between two or more different metamodels.

7.3.1 Anticipation Based on Structural Similarity in the Same Metamodel

In any given modeling language, a modeler could find components (i.e., classes, links, or attributes) that are "*structurally*" similar. For example, from the GRL metamodel presented in Figure 94, we can argue that class Contribution is structurally similar to class GRLLinkableElements in the sense that both have attributes related to *types* (i.e., ContributionType and ImportanceType, respectively) and *quantitative values* (i.e., quantitativeContribution and importanceQuantitative, respectively). In addition, constraint C6 in the Contribution class is semantically similar to constraint C8 in the GRLLinkableElements class (as shown in Figure 96) in that both constraints control the allowed number of datatypes (i.e., multiplicity) of attributes.

Having said that, if C6 is required to be relaxed (due to any delta introduced by a GRL model family) from quatitativeContribution:Integer=0 to quatitativeContribution:{Integer}={0}, (as discussed in Section 7.27.2.4), then we should anticipate C8 also to be relaxed (from importanceQuantitative:Integer=0 to importanceQuantitative:{Integer}= {0}) due to its structural similarity to C6.

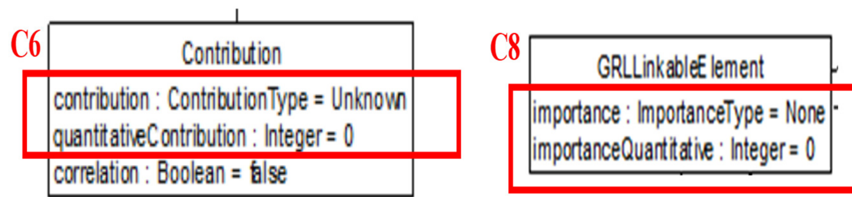


Figure 96 Example of structural similarity between classes in GRL's metamodel.

To generalize the above example for any potential model family of a given language, we suggest having two contingency matrices, one for the set of constraints (C_i) in MM with their associated relaxations (R_i), and the other for the similarity between constraints in a MM (based on the MM structure). Currently, similarity between constraints is decided manually by domain experts. We refer to the first matrix as the Constraints Relaxation Map (*ConReMap*, Table 10) and the other matrix as the Constraints Similarity Map (*ConSiMap*, Table 11).

Table 10 A constraint relaxation map (ConReMap).

	Constraints in MM				
Relaxations	C1	C2	C3	C4	C5
R1		√	?		?
R2	√		?	√	?
R3			?		?

Table 11 Constraint similarity map (ConSiMap) between MM constraints.

	Constraints in MM				
	C1	C2	C3	C4	C5
C1	—		√		
C2		—			√
C3	√		—		
C4				—	
C5		√			—

In Table 10, and based on the delta/constraints characterization (Section 7.2), let us assume that a family of models M1, M2, and M3 introduces violations to constraints C1, C2, and C4, where these constraints are relaxed according to relaxations R1 and R2. Up to this point, we do not know what kind of relaxation is needed (if any) for constraints C3 and C5. In addition, we have no idea if a new model (say M4), that could be added to the family, would introduce violations to any of the constraints (C1-C5).

To anticipate relaxation points in a particular metamodel (in order to support all potential model families), we suggest creating a constraint similarity matrix between the five constraints, as shown in Table 11 , and check if there are any syntactic and/or semantics similarities between them. From Table 11, let us assume that constraint C3 is similar to constraint C1, and constraint C5 is similar to constraint C2 (illustrated with√). From this similarity, we can infer that C3 should be relaxed in the same way as C1 was relaxed. That is, C3 would be relaxed using R2. Following the same rationale, C5 should be relaxed according to R1, since C5 is similar to C2, and C2 was already relaxed using R1 (as shown in Table 12 with underlined ticks). In this table, it can be noticed that relaxation R3 is not applied so far, and may not be needed.

Table 12 Anticipated relaxation points based on constraint similarity.

	Constraints in MM				
Relaxations	C1	C2	C3	C4	C5
R1		√			<u>√</u>
R2	√		<u>√</u>	√	
R3					

7.3.2 Anticipation Based on Structural Patterns across Different Metamodels

In addition to structural similarities recognized within one language's metamodel, a modeler might be interested in identifying structural similarities *across* many metamodels, such that relaxation decisions already applied to one language could be also adapted to other languages that share similar structures.

To identify structural commonalities across metamodels, we propose to borrow concepts from Cho and Gray's "*design patterns of metamodels*" domain [226] and apply them to our approach from a reverse-engineering point of view. Let us first present, briefly, the concept of metamodel design patterns, and then explain how this concept is adapted in our work.

Background: Design Patterns of Metamodels

The main purpose of metamodel design patterns is to provide design guidelines that help language designers (especially those who have little experience) produce high-quality metamodels and to provide consistent solutions for recurring metamodel design issues [226].

Design patterns of metamodels are inferred from the *common characteristics* of Domain Specific Modeling Languages (DSMLs), where the common characteristics represent reusable elements at a higher level of abstraction. For instance, based on their review and analysis of the concrete syntaxes of several DSMLs, Cho and Gray [226] derived a feature model that describes variabilities and commonalities of DSMLs. In that feature model, four major common features (i.e., Classifier, Relationship, Style, and Boundedness) are defined as mandatory features. There are two other optional features (i.e., Containment and Nesting) describing characteristics of a classifier. In addition, sub-features of Type, Orientation, and Boundedness are defined as alternatives because a relationship can have only one kind of Type, of Orientation, and of Boundedness.

To define design patterns for DSMLs, the authors assume that most modeling languages commonly use a *Box-and-Line* style, where boxes are generalized as a set of *Classifiers* and lines are mapped to *Relationships*. As a *Relationship* normally links two *Classifiers*, one for the source *Classifier* and the other for the target *Classifier*, the *Classifier* and *Relationship* are linked with two association relationships: *source* and *target*. *Multiplicity* is assigned to the association ends in order to specify the number of participating instances. From that generalization, the authors defined a metamodel design pattern that describes a very primitive concrete syntax, which consists of classifiers and association relationships (Figure 97.a).

To raise the level of abstraction, the authors also propose the containment design pattern (Figure 97.b) to represent the part-whole hierarchy by grouping large and complex model elements with a simple element called *container*. A metamodel class owning the

containment reference is called the *whole* or *container* and the metamodel class on the opposite side is called the *part* or *content*. A containment reference denotes an existential dependency (and cascading delete semantics) between the part and the whole.

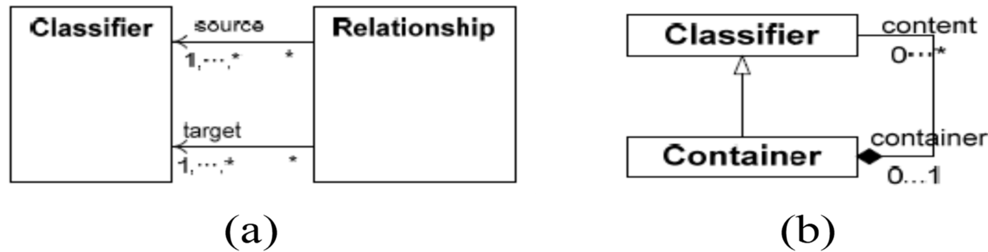


Figure 97 (a) Basic metamodel's components design pattern, and (b) containment design pattern, adapted from [226].

Adapting Generalizations from Metamodels' Design Patterns

In our work, we adapt the above concepts in a reverse engineering manner. We actually exploit the analysis of commonalities/variabilities already conducted on DSMLs, and benefit from the design patterns/idioms already defined by researchers in the domain of metamodels' design patterns. The purpose of using these foundations is to *generalize* the concrete syntax of different metamodels using common descriptors or labels. For example, if there are two classes (C1, C2) in a metamodel that follow the pattern illustrated in Figure 97.a, such that class C1 is linked with another class C2 by a source/target association links (with multiplicity), then we label or tag these classes using the original design patterns idioms (that were initially used to describe classes at the time of designing a metamodel). That is, C1 and C2 will be labeled as *Classifiers* and *Relationships,* respectively. Similarly, if two metamodel classes (C1, C2) in a metamodel follow the pattern illustrated in Figure 97.b, where C1 owns the containment reference (i.e., a container) and C2 is on the opposite side (i.e., content), then we label C1 as a *Container* and C2 as a *Classifier.*

After generalizing (or re-labelling) different metamodels with design pattern idioms, it becomes possible to recognize structural similarities (or *patterns*) across several metamodels. Such structural patterns, in turn, are used to provide key insights about how to apply the same relaxions on different metamodels.

To identify common structures between metamodels, we examined the concrete syntax of several metamodels, namely GRL, feature models, and state-transition diagrams

(STD). In particular, we analyzed commonalities between metamodels with specific focus on *classifiers*, *relationships*, and *containment*. Based on our analysis, we observe that despite the different syntaxes attached to classes and their relationships, different languages have structural patterns similar to those in Figure 97.

Figure 98 represents a simplified GRL metamodel with the main classes being relabelled with design pattern idioms. In this figure, classes Actor, IntElement and Link are labelled as *Container*, *Classifier*, and *Relationship*, respectively.

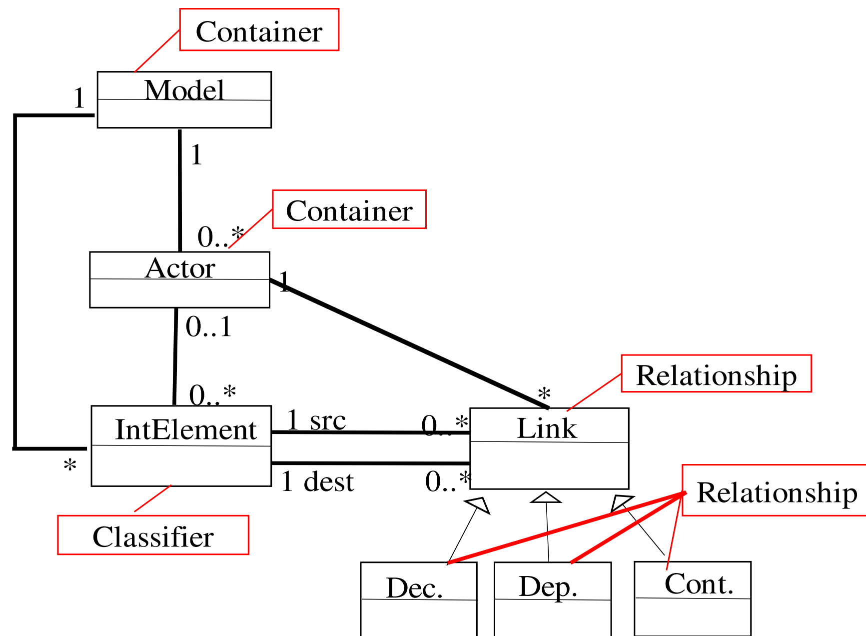


Figure 98 A simplified GRL metamodel tagged with general concepts.

Structural Patterns across different Metamodels

Referring to the tagged metamodel in Figure 98, we can notice the following structural patterns, P_i (illustrated in Figure 99): a Classifier-Relationship pattern (*P1*), a Container-Classifier pattern (*P2*), a Container-Container pattern (*P3*) and a Container-Relationship pattern (*P4*).

Analyzing the evolution scenarios of GRL model families (and based on the usage pattern of the GRL language), we have already discussed in [185][186] that the evolution/variation of models in a GRL model family required relaxing the multiplicity constraint of the association ends between "IntentionalElements" and "Links" (i.e., constraints in *P1*) from *1 src* to *1..* src* and from *1 dest* to *1..* dest*. In addition, the multiplicity

constraint in *P2* needed to be relaxed at the Actor side from *0..1* to *0..** in the constraints related to "Actors" with "IntentionalElements". Note that *P3* and *P4* did not need further relaxion.

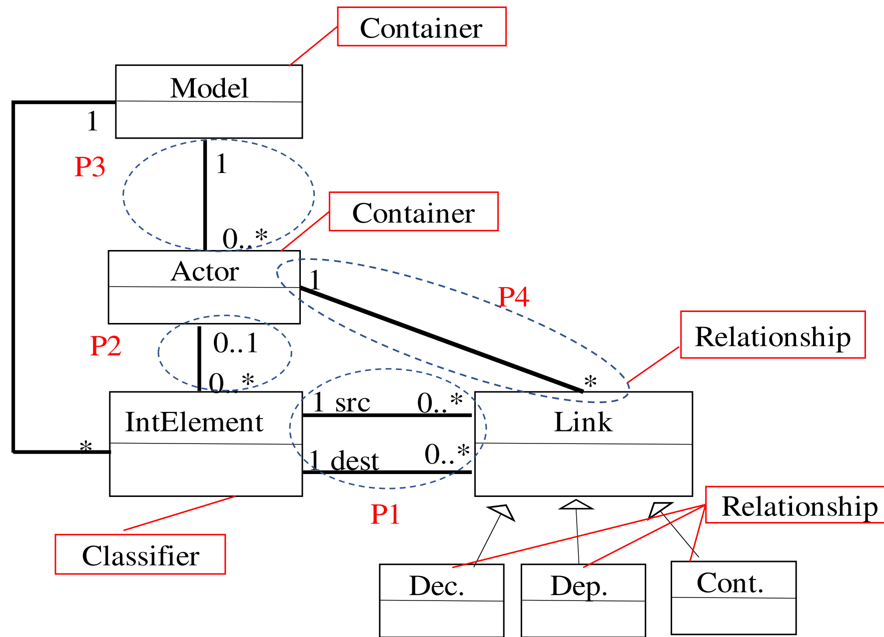
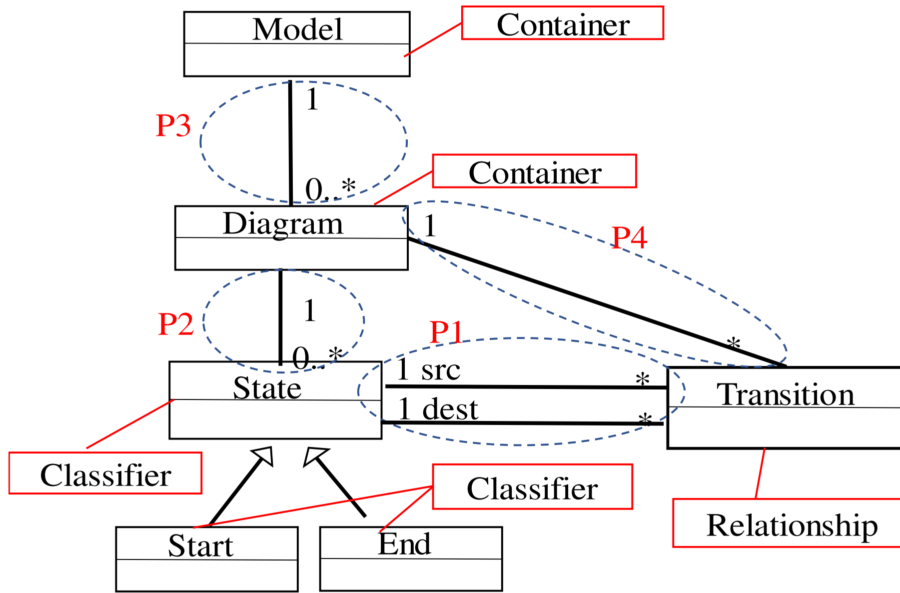


Figure 99 Structural patterns in GRL metamodel.

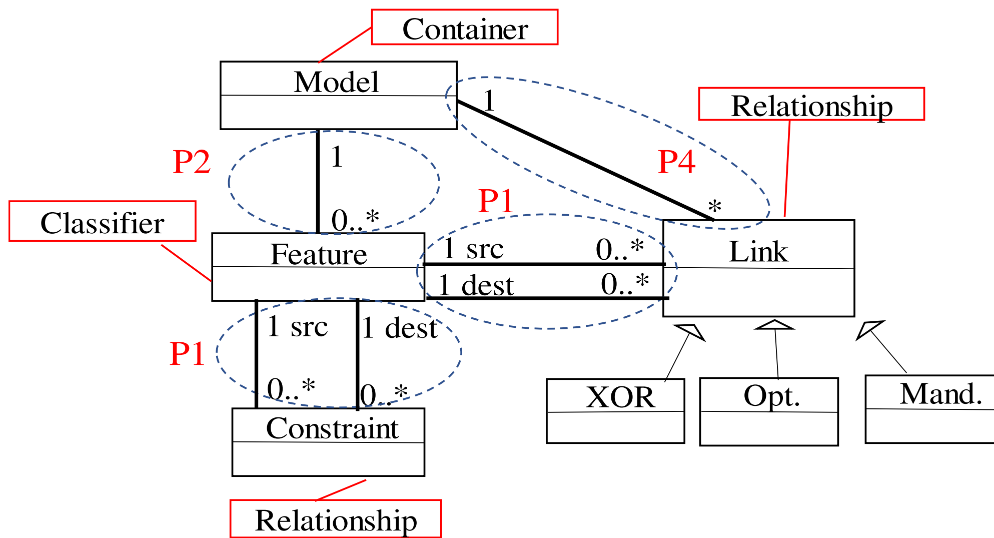
The same patterns illustrated in Figure 99 can be found in other languages metamodels, such as those of state transition diagrams (STD) and feature models (FM), as shown in Figure 100.

Anticipating Relaxations Across Metamodels

Having similar patterns across different languages can help anticipating required relaxations in one language based on relaxations already found in another language. For instance, *P1* in GRL's metamodel is similar to both *P1* in STD's and *P1* in FM's. If a relaxation is performed for multiplicity constraints of *P1* in GRL, then the same relaxation would be performed for multiplicity constraints of *P1* in the STD and FM metamodels. The same thing applies to *P2*. For *P3* and *P4*, no relaxation would be needed.



(a) STD Metamodel



(b) FM Metamodel

Figure 100 Structural patterns in (a): STD metamodel, and (b): FM metamodel

7.3.3 Discussion

After introducing the two solutions proposed for inferring metamodel relaxation points, it is important to discuss when to use solution 1 (i.e., the Evi-MeReFam) and when to use

solution 2 (i.e., the Anti-MeReFam). In general, there are no obligatory rules about using solution 1 before solution 2, or the other way around. However, solution 1 is usually suggested whenever we are dealing with a particular modeling language for the first time, where there is still no previous evidence about the usage pattern of that language or about the evolution/change behavior of current model families of this language. In this scenario, the suggested relaxation points are related directly to the types of delta for a particular modeling language and for specific models in a family.

Building enough evidence about relaxation points in a metamodel would help a modeler to anticipate other relaxation points based on the structure of the language, independently of models in a family. This is where solution 2 takes place. However, a modeler can still start directly with solution 2 without relying on the pure evidence-based solution (i.e., solution 1). In this case, a modeler needs to examine a metamodel to infer its structural patterns, thereafter she would be able to anticipate where relaxations are needed based on the structure and also usage patterns of the language, as discussed in Sections 7.2.

7.4. Chapter Summary

In this chapter, we discussed the challenge of capturing model families using union models, where the later may not conform to the same metamodel that the individual models conform to. The non-conformance issues stem mainly from the fact that a union model could contain links or attributes that violate a metamodel's internal multiplicity constraints (related to attributes and association ends) or other external constraints (in OCL). To support the representation of union models as valid instances of a language's metamodel, we propose to relax that metamodel, *minimally*, such that the upper bound of only particular multiplicity constraints of association ends and/or attributes are relaxed. We proposed two heuristic solutions for metamodel relaxations. The first solution is the Evi-MeReFam approach, which supports metamodel relaxation based on evidence provided by the usage pattern of a language and the evolution/change behavior of current model families of that language. The second solution is the Anti-MeReFam, which anticipates metamodel relaxation points based on structural patterns to support potential (prospective) model families of a given language, or across multiple languages. The purpose of inferring only specific

relaxation points in a metamodel is to be able to adapt the existing tools and analysis techniques once and minimally for all potential model families of a given modeling language.

For future work, we plan to work on the discovery of a catalogue of relaxation patterns for metamodels and their empirical assessment. We also need to extend this approach to cover external constraints (e.g., in OCL).

Chapter 8. Conclusions and Future Work

This chapter recalls the thesis contributions and relates them to the five research questions (RQ1-RQ5) found in Section 1.5. In addition, it re-emphasizes the differences between these contributions and closely-related work from the literature. The chapter also discusses threats to the validity of the findings as well as future directions and research opportunities.

8.1. Summary and Contributions

In Model-Driven Engineering, models change continuously to reflect new requirements or refined understanding about the domain, resulting in a so-called *model families*. For a given modeling language, a model family is a set of related models, with commonalities and variabilities, that result from the evolution of models into *several versions over time and/or the variation of models over the space dimension*. In contexts where there are several versions/variations of a model, analyzing individual models, *one model at a time*, becomes cumbersome and inefficient, especially when many models share several elements in common (i.e., redundancy). In addition, other kinds of analyses that rely on temporal or spatial information (e.g., trend analysis over time) become complex using individual models, separately.

This thesis proposed *union models* to: (1) support the representation of model families for time and space dimensions, (2) achieve performance gains during analysis of family models all at once, compared to the analysis of individual models, one model at a time, and (3) support types of analyses that are more easily feasible with union models compared with individual models. The **contributions** of the thesis are:

- A language-independent, graph theory-based formalization of model families and union models (RQ1), defined in Chapter 4.
- A language independent algorithm (based on our formalization) to produce a union model from a set of models (in a compact and exact manner) of a given language (RQ1), with an appropriate predicate encoding (PELA) used as a concrete syntax, also presented in Chapter 4.

- A Spatio-Temporal Annotation Language (STAL) to support the representation of variability in model families (in space and time dimensions) and to facilitate reasoning about union models, together with a merging algorithm (RQ1), presented in Chapter 3.
- Improved efficiency of analysis and reasoning over a set of models, all at once (using the union model) compared to reasoning on single models, one model at a time (RQ2). Chapter 5 reports on three language-independent reasoning tasks (applied to families of state machines and GRL goal models) whereas Chapter 6 reports on language-specific reasoning (in GRL).
- Examples of analysis techniques adapted to support efficient reasoning about model families (RQ3), with a focus on forward and backward evaluation techniques for GRL models, in Chapter 6.
- An implementation of the algorithm used for creating a union model from individual GRL models, with a translation to CPLEX for supporting forward and backward propagation in families of goal models (Chapter 6).

The thesis also provides the following minor contributions, both presented in Chapter 7:

- This thesis addresses the metamodel-level challenges associated with union models. In particular, it contributes a characterization of the requirements for minimally relaxing modeling languages (based on the structure of original metamodels) to support all potential union models of a language (RQ4).
- The thesis proposes two methods (*Evi-MeReFam* and *Anti-MeReFam*) to infer/anticipate locations where metamodel relaxations are needed (i.e., relaxation points) so that existing tools and analysis techniques be adapted *once* per language (RQ5).

8.2. Differences between this Thesis and Closely-Related Work

This thesis identified the gaps that exist in current approaches (Chapter 2) and illustrated how this thesis' work is different. These differences can be summarized as follows:

- The annotation mechanisms used in the literature are different from our proposed annotations. To the best of our knowledge, the spatio-temporal annotation language

that we use to annotate elements of union models with information about versions/configurations has never been proposed in the literature.

- None of the approaches discussed in Sections 2.3 to 2.6 consider the evolution of models over time and the variation over the space dimension in the way that we handle it (i.e., using one single, annotated union model). In particular, SPL-related approaches focus on modeling variability over the space/product dimension. VC-related approaches focus on handling and managing variability of models over the time dimension. Even the approaches that handle variability in space and time together (Section 2.6) address the problem from the perspective of SPL, which is different from our approach.
- Unlike the use of union models (which does not require feature models), all SPL-related approaches (Section 2.3 and Section 2.5) rely heavily on the existence of feature models to construct 150% models and to extract individual models from them.
- The usage of union model-like artifacts (mainly the 150% models), as discussed in Section 2.7, does not go beyond the ability to extract products to conduct only particular and traditional types of analysis (such as testing, model checking, typed checking, or theorem proving). However, our method exploits union models (with annotations) to perform more sophisticated kinds of analysis, such as trend analysis.
- The use of goal models to manage variability (as discussed in Section 2.8) did not consider variability of models along the time dimension.

At the metamodel level, this thesis revealed a lack of approaches that deal with metamodel co-evolution in light of model evolution in the general context (Figure 14) and in the context of model families in particular (Figure 6), which is the typical problem discussed in this thesis. The gaps between the existing research and our proposed research are discussed in Sections 2.10 to 2.15 and summarized as follows:

- Most approaches conduct transformation/migration of models each time a metamodel changes (Section 2.10); this requires a non-trivial adaptation effort for tools and analysis techniques. Our proposed approach (in Chapter 7), on the other hand,

aims to infer a single relaxed metamodel that accommodates all potential union models of a language, so as to develop tools for this relaxed metamodel only *once*.

- The driving factor of evolution is different. While existing approaches deal with models and/or operations co-evolution triggered by metamodel evolution, our work targets the evolution/relaxation of metamodels triggered by model evolution in the context of model families (Section 2.10 and Section 2.12).
- Some of the related approaches (Section 2.11) either add new concepts to the original metamodel or modify the language's validity constraints by further constraining their restrictions, whereas our approach only intends to relax some constraints (as new concepts are not required).
- All the approaches discussed in Section 2.13 and Section 2.14 propose to relax *all* constraints/restrictions that exist in a metamodel (including invariants and mandatory features, and by enforcing the nonexistence of instances of certain class). This extreme relaxation would have a negative impact on tools adaptation. Our proposed approach, on the other hand, is different in the sense that it aims to relax particular constraints related to multiplicities of attributes and association ends, together with OCL-like constraints, in a minimal way.
- Finally, the context in which a metamodel is relaxed and the purpose of relaxing metamodels in all existing approaches is completely different than the purpose of our proposed relaxation method (which is mainly to support the representation of model families, not to generate new types of products).

8.3. Threats to Validity

In our work, there are several threats to validity that must be assessed to better characterize the limitations of the approach and results presented in this thesis. According to Perry et al. [229], the following are the most relevant categories of threats to validity for our work.

8.3.1 Construct Validity

Construct validity aims to assess to which extent the experiments, tests and/or case studies actually answer our research hypothesis. An important threat in this thesis is that the chosen

examples may not reflect the complexity of real environments. Another important issue here is the unavailability of several real models for the construction of union models, and the reliance on generated models (more convenient for performance evaluations). Although the size of models and of families was taken into consideration during experiments, other characteristics such as the model topologies (e.g., different depths, proportion of overlap) and the complexity of STAL annotations were not. In addition, the complexity of transforming metamodels and models into the formal graph-based representation was not taken into consideration. Finally, the usability of the approach (based on the experience of others) was not assessed.

8.3.2 Internal Validity

Internal validity examines any bias and other confounding factors, and asses the degree to which conclusions about causal relationships can be made based on the test settings and measures obtained. One obvious threat in this thesis is that bias might be introduced by having the thesis author perform the tests, collect the raw data, and analyze the results. Similarly, the selection of languages and analysis techniques in the experiments (mainly based on convenience and local expertise) might have been biased towards options for which union models are favorable. A similar issue pertains to the literature review, mainly done by a single person under the supervision of another one.

8.3.3 External Validity

External validity verifies the extent to which the evaluation results can be generalized to other cases or contexts. One important external threat encountered here is that the experiment data target two modeling languages only (GRL and state machine diagrams), and a handful of language-independent and language dependent analysis techniques. Other languages and analysis techniques may not result in efficiency improvements as positive as the ones observed here.

8.4. Future Work

There are many opportunities to follow up on the thesis work, including the following directions:

- **Study the effects of variation and topology on reasoning techniques**: we need to describe how sensitive reasoning and analysis are to the degree of variation in a union model. The degree of variation of an M_U can be inferred from the number of annotations and the number of annotated elements that exist in a model family and represented by that M_U. In other words, a union model is considered as highly variable if it has on average a large number of annotations per element. The topology of individual models and of the resulting union model (in terms of maximum/average depth, maximum/average number of connections per node, etc.) also deserved to be further studied to better characterize the potential for performance improvements.
- **Investigate other potential analysis techniques for model families**: inspired by the data analytics domain [228], we envision conducting several types of analyses over a model family to reason on all members of the family, which otherwise would not be feasible to be conducted on individual models. Such analysis approaches include:
  1. Descriptive analysis to describe the main aspects or features of the models being analyzed. This allows one to make comparisons among several models in a family;
  2. Exploratory analysis to analyze models to find previously unknown relationships. This type of analysis is useful for discovering new connections and to provide future recommendations;
  3. Inference analysis to infer information about a large population of models based on a small sample of models. For example, examining compliance levels using a small sample of models to explain how well the entire system complies with regulations and rules;
  4. Predictive analysis to analyze current and historical (or legacy) models to predict future occurrences of events. The essence of such approaches is to use data on some models to predict values for other models; and
  5. Causal analysis to figure out what will happen to one or more models when some model gets changed. For instance, to study the impact of changing one or more rule in a particular regulatory model on the behavior of other models in the family.

- **Characterization of the languages and analysis techniques that may benefit from union models**: although the results reported in this thesis are highly positive, we suspect this might not always be the case depending on the language or particular analysis technique being used. A better understanding of where union models can bring value and where they cannot (prior to implementation) would be beneficial. Moreover, for the language-specific analysis (as proposed in Chapter 6), it will be beneficial to provide guidelines on how to lift analysis techniques for another language, e.g., BPMN or UCM model families, and simulations or traversals. This process requires studying and understanding the semantics of languages prior to adapting or lifting their existing analysis techniques to model families.
- **Validation of the Evi-MeReFam and Anti-MeReFam approaches**: the empirical validation of the two proposed relaxation approaches (in isolation and in combination) could be further studied using the Desmet method from Kitchenham et al. [227]. The applicability of Evi-MeReFam and Anti-MeReFam can be evaluated empirically, based on a collection of models in different languages (e.g., GRL, state transition diagrams, and feature models), so the results are not language dependent. The experiments could involve having several model families that are either synthetic (i.e., auto generated by tools) or real (from domains with highly evolving/varying models, such as regulatory models, or SPL models, to which we have access). *Precision* and *recall* can be used as two metrics to measure the accuracy of our approaches. The outcome would represent the beginning of a catalogue of relaxation patterns for metamodels.
- **Improve tool support**: although a prototype tool exists for creating union models, the software ecosystem could be greatly enhanced by providing converters from metamodels (in EMF or MOF) to type graphs and models to typed graphs expressed with PELA. Tools to better visualize annotated union models in the original language syntax would also be useful. Support for more rigorously lifting analysis methods and editors would also contribute to the adoption in practice of the proposed approach. Optimizations of current implementations are also possible. For instance, STAL annotations might be more efficiently stored and accessed with indexed lookup tables.

- **Study the usability of the approach**: there is a need to assess whether the approach is useful to practitioners, and which parts (especially regarding automation) require usability improvements to improve chances of adoption in industry.

References

[1] Paige, R. F., Kolovos, D. S., & Polack, F. A. (2014). A tutorial on metamodelling for grammar researchers. *Science of Computer Programming*, *96*, 396-416.

[2] Object Management Group, *Object Constraint Language (OCL)*, Version 2.4. Needham, MA, 2014.

[3] Pohl, K., Böckle, G., & van Der Linden, F. J. (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[4] Schobbens, P. Y., Heymans, P., Trigaux, J. C., & Bontemps, Y. (2007). Generic semantics of feature diagrams. *Computer Networks*, *51*(2), 456-479.

[5] Micouin, P. (2014). Model Based Systems Engineering: Fundamentals and Methods. John Wiley & Sons.

[6] International Telecommunication Union (2018). *Recommendation Z.151 (10/18) User Requirements Notation (URN) - Language definition*. October 2018. Online: https://www.itu.int/rec/T-REC-Z.151/en. (accessed October 2019).

[7] Amyot, D., & Mussbacher, G. (2011). User Requirements Notation: the first ten years, the next ten years. *JSW*, *6*(5), 747-768.

[8] Palmieri, A., Collet, P., & Amyot, D. (2015). Handling regulatory goal model families as software product lines. In *International Conference on Advanced Information Systems Engineering* (pp. 181-196). Springer, Cham.

[9] Shamsaei, A., Amyot, D., Pourshahid, A., Braun, E., Yu, E., Mussbacher, G., Tawhid, R., & Cartwright, N. (2012). An approach to specify and analyze goal model families. In *International Workshop on System Analysis and Modeling* (pp. 34-52). Springer, Berlin, Heidelberg.

[10] Famelis, M., Rubin, J., Czarnecki, K., Salay, R., & Chechik, M. (2017). Software Product Lines with Design Choices: Reasoning about Variability and Design Uncertainty. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 93-100). IEEE.

[11] Seidl, C., Schaefer, I., & Aßmann, U. (2014). Integrated management of variability in space and time in software families. In *Proceedings of the 18th International Software Product Line Conference* (Vol 1, pp. 22-31). ACM.

[12] Lity, S., Nahrendorf, S., Thüm, T., Seidl, C., & Schaefer, I. (2018). 175% modeling for product-line evolution of domain artifacts. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems* (pp. 27-34). ACM.

[13] March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision support systems*, *15*(4), 251-266.

[14] Von Alan, R. H., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS quarterly*, *28*(1), 75-105.

[15] Stahl, T., Voelter, M., & Czarnecki, K. (2006). *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc.

[16] Van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, *35*(6), 26-36.

[17] Object Management Group (OMG). *Meta Object Facility (MOF)* Version 2.5.1. Needham, MA, 2016.

[18] Istoan, P., Klein, J., Perrouin, G., & Jézéquel, J. M. (2011). A metamodel-based classification of variability modeling approaches. In *VARY, International Workshop affiliated with ACM/IEEE 14th International Conference on Driven Engineering Languages and Systems*. IT University of Copenhagen.

[19] Karagiannis, D., & Kühn, H. (2002). Metamodelling platform. *In Proceedings of third International Conference EC-Web 2002*. LNCS, (Vol. 2455, pp. 182-196). Springer, Heidelberg.

[20] Northrop, L., Clements, P., Bachmann, F., Bergey, J., Chastek, G., Cohen, S., Donohoe, P., Jones, L., Krut, R., & Little, R. (2007). *A framework for software product line practice*, version 5.0. SEI, Carnegie-Mellon University, USA.

[21] Van der Linden, F. (2002). Software product families in Europe: the Esaps & Café projects. *IEEE software*, *19*(4), 41-49.

[22] Halmans, G., & Pohl, K. (2003). Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, *2*(1), 15-36.

[23] Northrop, L. M. (2002). SEI's software product line tenets. *IEEE software*, *19*(4), 32-40.

[24] Coplien, J., Hoffman, D., & Weiss, D. (1998). Commonality and variability in software engineering. *IEEE software*, *15*(6), 37-45.

[25] Alves, V., Niu, N., Alves, C., & Valença, G. (2010). Requirements engineering for software product lines: A systematic literature review. *Information and Software Technology*, 52(8), 806-820.

[26] Fitzpatrick, B. W., Pilato, C. M., & Collins-Sussman, B. (2004). *Version control with Subversion*. O'Reilly Media.

[27] Gomaa, H. (2006). Designing software product lines with UML 2.0: From use cases to pattern-based software architectures. In *Proceedings of the 9th international conference on Reuse of Off-the-Shelf Components* (pp. 440-440). Springer-Verlag.

[28] Jézéquel, J. M. (2012). Model-driven engineering for software product lines. *ISRN Software Engineering*, *2012*.

[29] Bachmann, F., & Clements, P. C. (2005). *Variability in software product lines* (No. CMU/SEI-2005-TR-012). Carnegie-Mellon University, Pittsburgh, USA.

[30] Weiss, D. M., & Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process* (Vol. 12). Reading: Addison-Wesley.

[31] Halmans, G., & Pohl, K. (2003). Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, *2*(1), 15-36.

[32] Niu, N., Savolainen, J., & Yu, Y. (2010). Variability modeling for product line viewpoints integration. In *2010 IEEE 34th Annual Computer Software and Applications Conference* (pp. 337-346). IEEE.

[33] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). *Feature-oriented domain analysis (FODA) feasibility study* (No. CMU/SEI-90-TR-21). Carnegie-Mellon University, Pittsburgh, USA.

[34] Czarnecki, K. (2002). Generative programming: Methods, techniques, and applications tutorial abstract. In *International Conference on Software Reuse* (pp. 351-352). Springer, Berlin, Heidelberg.

[35] Griss, M. L. (2000). Implementing product-line features with component reuse. In *International Conference on Software Reuse* (pp. 137-152). Springer, Berlin, Heidelberg.

[36] Czarnecki, K., & Antkiewicz, M. (2005). Mapping features to models: A template approach based on superimposed variants. In *International conference on generative programming and component engineering* (pp. 422-437). Springer, Berlin, Heidelberg.

[37] Schaefer, I. (2010). Variability Modelling for Model-Driven Development of Software Product Lines. In *Proceedings of the Fourth International Workshop on Variability Modeling of Software-Intensive Systems*, (Vol. 10, pp. 85-92). ACM.

[38] Ziadi, T., Hélouët, L., & Jézéquel, J. M. (2003). Towards a UML profile for software product lines. In *International Workshop on Software Product-Family Engineering* (pp. 129-139). Springer, Berlin, Heidelberg.

[39] Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G. K., & Svendsen, A. (2008). Adding standardized variability to domain specific languages. In *2008 12th International Software Product Line Conference* (pp. 139-148). IEEE.

[40] Object Management Group (OMG). *Common Variability Language (CVL) Revised Submission*. Needham, MA, 2012.

[41] Bąk, K., Czarnecki, K., & Wąsowski, A. (2010). Feature and meta-models in Clafer: mixed, specialized, and coupled. In *International Conference on Software Language Engineering* (pp. 102-122). Springer, Berlin, Heidelberg.

[42] Buchmann, T., Schwägerl, F., Störrle, H., Botterweck, G., Bourdells, M., Kolovos, D., & Paige, R. (2012). FAMILE: tool support for evolving model-driven product lines. In *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications,* CEUR WS (pp. 59-62).

[43] Heidenreich, F., Kopcsek, J., & Wende, C. (2008). FeatureMapper: mapping features to models. In *Companion of the 30th international conference on Software engineering* (pp. 943-944). ACM.

[44] Loughran, N., Sánchez, P., Garcia, A., & Fuentes, L. (2008). Language support for managing variability in architectural models. In *International Conference on Software Composition* (pp. 36-51). Springer, Berlin, Heidelberg.

[45] Apel, S., Janda, F., Trujillo, S., & Kästner, C. (2009). Model superimposition in software product lines. In *International Conference on Theory and Practice of Model Transformations* (pp. 4-19). Springer, Berlin, Heidelberg.

[46] Clarke, D., Helvensteijn, M., & Schaefer, I. (2010). Abstract delta modeling. *Proc. 9th International Conference on Generative Programming and Component Eng.*, (pp. 13-22).

[47] Schaefer, I., Bettini, L., Bono, V., Damiani, F., & Tanzarella, N. (2010). Delta-oriented programming of software product lines. In *International Conference on Software Product Lines* (pp. 77-91). Springer, Berlin, Heidelberg.

[48] Lackner, H., & Schmidt, M. (2015). Potential Errors and Test Assessment in Software Product Line Engineering. *arXiv preprint arXiv:1504.02443*.

[49] Object Management Group (OMG) (2015). *XML Metadata Interchange (XMI),* Version 2.5.1. Needham, MA, 2015.

[50] Altmanninger, K., Seidl, M., & Wimmer, M. (2009). A survey on model versioning approaches. *International Journal of Web Information Systems*, *5*(3), 271-304.

[51] Koegel, M., & Helming, J. (2010). EMFStore: a model repository for EMF models. In *2010 ACM/IEEE 32nd International Conference on Software Engineering* (Vol. 2, pp. 307-308). IEEE.

[52] Murta, L., Corrêa, C., Prudêncio, J. G., & Werner, C. (2008). Towards odyssey-VCS 2: improvements over a UML-based version control system. In *Proceedings of the 2008 international workshop on Comparison and versioning of software models* (pp. 25-30). ACM.

[53] Schwägerl, F., & Westfechtel, B. (2017). Perspectives on combining model-driven engineering, software product line engineering, and version control. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems* (pp. 76-83). ACM.

[54] Schneider, C., Zündorf, A., & Niere, J. (2004). CoObRA-a small step for development tools to collaborative environments. *In Workshop on Directions in Software Engineering Environments*.

[55] Brun, C., & Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, *9*(2), 29-34.

[56] Kelter, U., Wehren, J., & Niere, J. (2005). A Generic Difference Algorithm for UML Models. *Software Engineering*, *64*(105-116), 4-9.

[57] Kehrer, T., Kelter, U., & Taentzer, G. (2013). Consistency-preserving edit scripts in model versioning. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 191-201). IEEE.

[58] Conradi, R., & Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Surveys (CSUR)*, *30*(2), 232-282.

[59] Westfechtel, B. (2014). Merging of EMF models. *Software & Systems Modeling*, *13*(2), 757-788.

[60] Schwägerl, F., Uhrig, S., & Westfechtel, B. (2013). Model-based tool support for consistent three-way merging of EMF models. In *Proceedings of the workshop on ACadeMics Tooling with Eclipse* (p. 2). ACM.

[61] Lippe, E., & Van Oosterom, N. (1992). Operation-based merging. In *ACM SIGSOFT Software Engineering Notes* (Vol. 17, No. 5, pp. 78-87). ACM.

[62] Alanen, M., & Porres, I. (2003). Difference and union of models. In *International Conference on the Unified Modeling Language* (pp. 2-17). Springer, Berlin, Heidelberg.

[63] Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W., & Wimmer, M. (2008). AMOR–towards adaptable model versioning. In *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS* (Vol. 8, pp. 4-50).

[64] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., & Sabetzadeh, M. (2006). A manifesto for model merging. In *Proceedings of the 2006 international workshop on Global integrated model management* (pp. 5-12). ACM.

[65] Xing, Z., & Stroulia, E. (2005). UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 54-65). ACM.

[66] Richards, D. (2003). Merging individual conceptual models of requirements. *Requirements Engineering*, *8*(4), 195-205.

[67] Sabetzadeh, M., & Easterbrook, S. (2005). An algebraic framework for merging incomplete and inconsistent views. In *13th IEEE International Conference on Requirements Engineering (RE'05)* (pp. 306-315). IEEE.

[68] Niu, N., Easterbrook, S., & Sabetzadeh, M. (2005). A category-theoretic approach to syntactic software merging. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (pp. 197-206). IEEE.

[69] Easterbrook, S., & Chechik, M. (2001). A framework for multi-valued reasoning over inconsistent viewpoints. In *Proceedings of the 23rd international conference on software engineering* (pp. 411-420). IEEE Computer Society.

[70] Uchitel, S., & Chechik, M. (2004). Merging partial behavioural models. In *ACM SIGSOFT Software Engineering Notes* (Vol. 29, No. 6, pp. 43-52). ACM.

[71] Schmid, K., Koschke, R., Kröher, C., & Lüdemann, D. (2013). Towards identifying evolution smells in Software Product Lines. *Software Engineering 2013-Workshopband*.

[72] Seidl, C., Schaefer, I., & Aßmann, U. (2014). Capturing variability in space and time with hyper feature models. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems* (p. 6). ACM.

[73] Seidl, C., Schaefer, I., & Aßmann, U. (2014). Integrated management of variability in space and time in software families. In *Proceedings of the 18th International Software Product Line Conference,* (pp. 22-31). ACM.

[74] Mitschke, R., & Eichberg, M. (2008). Supporting the evolution of software product lines. In *ECMDA Traceability Workshop (ECMDA-TW)* (pp. 87-96).

[75] Czarnecki, K., Helsen, S., & Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, *10*(1), 7-29.

[76] Dhungana, D., Neumayer, T., Grunbacher, P., & Rabiser, R. (2008). Supporting evolution in model-based product line engineering. In *2008 12th International Software Product Line Conference* (pp. 319-328). IEEE.

[77] Font, J., Ballarín, M., Haugen, Ø., & Cetina, C. (2015). Automating the variability formalization of a model family by means of common variability language. In *Proceedings of the 19th International Conference on Software Product Line* (pp. 411-418). ACM.

[78] Avila-García, O., García, A. E., & Rebull, E. (2007). Using software product lines to manage model families in model-driven engineering. In *Proceedings of the 2007 ACM symposium on Applied computing* (pp. 1006-1011). ACM.

[79] Rebull, E. V. S., Avila-Garcıa, O., Garcıa, J. L. R., & Garcıa, A. E. (2007). Applying a model driven approach to model transformation development. *In MDEIS'2007: Proceedings of the III International Workshop on Model-Driven Enterprise Information Systems*.

[80] Martinez, J., Ziadi, T., Klein, J., & Le Traon, Y. (2014). Identifying and visualising commonality and variability in model variants. In *European Conference on Modelling Foundations and Applications* (pp. 117-131). Springer, Cham.

[81] Polzer, A., Merschen, D., Botterweck, G., Pleuss, A., Thomas, J., Hedenetz, B., & Kowalewski, S. (2012). Managing complexity and variability of a model-based embedded software product line. *Innovations in Systems and Software Engineering*, *8*(1), 35-49.

[82] Grönniger, H., Krahn, H., Pinkernell, C., & Rumpe, B. (2008). Modeling variants of automotive systems using views. In *Proceedings of Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)* (pp. 76-89).

[83] Weißleder, S., & Lackner, H. (2013). Top-down and bottom-up approach for model-based testing of product lines. In *MBT*, pp. 82–94, 2013.

[84] Cichos, H., Oster, S., Lochau, M., & Schürr, A. (2011). Model-based coverage-driven test suite generation for software product lines. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 425-439). Springer, Berlin, Heidelberg.

[85] Oster, S., Zorcic, I., Markert, F., & Lochau, M. (2011). MoSo-PoLiTe: tool support for pairwise and model-based software product line testing. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems* (pp. 79-82). ACM.

[86] Lapouchnian, A., & Mylopoulos, J. (2009). Modeling domain variability in requirements engineering with contexts. In *International Conference on Conceptual Modeling* (pp. 115-130). Springer, Berlin, Heidelberg.

[87] Lapouchnian, A., & Mylopoulos, J. (2011). Capturing Contextual Variability in i* Models. In *IStar Workshop* (pp. 96-101). CEUR, Vol. 766.

[88] Ali, R., Dalpiaz, F., & Giorgini, P. (2010). A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, *15*(4), 439-458.

[89] Silva, C. T., Borba, C., & Castro, J. (2011). A Goal Oriented Approach to Identify and Configure Feature Models for Software Product Lines. In *WER*.

[90] Borba, C., & Silva, C. (2009). A comparison of goal-oriented approaches to model software product lines variability. In *International Conference on Conceptual Modeling* (pp. 244-253). Springer, Berlin, Heidelberg.

[91] Mussbacher, G., Araújo, J., Moreira, A., & Amyot, D. (2012). AoURN-based modeling and analysis of software product lines. *Software Quality Journal*, *20*(3-4), 645-687.

[92] Yu, Y., do Prado Leite, J.C.S., Lapouchnian, A., & Mylopoulos, J. (2008). Configuring features with stakeholder goals. In *Proceedings of the 2008 ACM symposium on Applied computing* (pp. 645-649). ACM.

[93] Aprajita, S.L., & Mussbacher, G. (2017). Specifying evolving requirements models with TimedURN. In *Proceedings of the 9th International Workshop on Modelling in Software Engineering* (pp. 26-32). IEEE Press.

[94] Mussbacher, G. (2016). TimedGRL: Specifying Goal Models Over Time. In *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)* (pp. 125-134). IEEE.

[95] Grubb, A. M., & Chechik, M. (2016). Looking into the crystal ball: requirements evolution over time. In *2016 IEEE 24th International Requirements Engineering Conference (RE)* (pp. 86-95). IEEE.

[96] Thüm, T., Apel, S., Kästner, C., Schaefer, I., & Saake, G. (2014). A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, *47*(1), 1-45.

[97] Oster, S., Markert, F., & Ritter, P. (2010). Automated incremental pairwise testing of software product lines. In *International Conference on Software Product Lines* (pp. 196–210). Springer, Berlin, Heidelberg.

[98] Perrouin, G., Sen, S., Klein, J., Baudry, B., & Le Traon, Y. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In *2010 Third international conference on software testing, verification and validation* (pp. 459-468). IEEE.

[99] Apel, S., Kästner, C., & Lengauer, C. (2008). Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th international conference on Generative programming and component engineering* (pp. 101-112). ACM.

[100] Istoan, P. (2013). *Methodology for the derivation of product behaviour in a Software Product Line* (Doctoral dissertation), Université de Rennes 1, France.

[101] Buchmann, T., & Schwägerl, F. (2012). Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development* (pp. 37-44). ACM.

[102] Katz, S. (2006). Aspect categories and classes of temporal properties. In *Transactions on aspect-oriented software development I* (pp. 106-134). Springer, Berlin, Heidelberg.

[103] Cordy, M., Schobbens, P. Y., Heymans, P., & Legay, A. (2012). Towards an incremental automata-based approach for software product-line model checking. In *Proceedings of the 16th International Software Product Line Conference,* (pp. 74-81). ACM.

[104] Fantechi, A., & Gnesi, S. (2008). Formal modeling for product families engineering. In *2008 12th International Software Product Line Conference* (pp. 193-202). IEEE.

[105] Bruns, D., Klebanov, V., & Schaefer, I. (2010). Verification of software product lines with delta-oriented slicing. In *International Conference on Formal Verification of Object-Oriented Software* (pp. 61-75). Springer, Berlin, Heidelberg.

[106] Thaker, S., Batory, D., Kitchin, D., & Cook, W. (2007). Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering* (pp. 95-104). ACM.

[107] Apel, S., Kästner, C., Größlinger, A., & Lengauer, C. (2010). Type safety for feature-oriented product lines. *Automated Software Engineering*, *17*(3), 251-300.

[108] Kolesnikov, S., von Rhein, A., Hunsen, C., & Apel, S. (2013). A comparison of product-based, feature-based, and family-based type checking. In *ACM SIGPLAN Notices* (Vol. 49, No. 3, pp. 115-124). ACM.

[109] Ribeiro, M., Pacheco, H., Teixeira, L., & Borba, P. (2010). Emergent feature modularization. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (pp. 11-18). ACM.

[110] Tartler, R., Lohmann, D., Sincero, J., & Schröder-Preikschat, W. (2011). Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *Proceedings of the sixth conference on Computer systems* (pp. 47-60). ACM.

[111] Adelsberger, S., Sobernig, S., & Neumann, G. (2014). Towards assessing the complexity of object migration in dynamic, feature-oriented software product lines.

In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems* (p.17). ACM.

[112] Sabouri, H., & Khosravi, R. (2014). Reducing the verification cost of evolving product families using static analysis techniques. *Science of Computer Programming*, *83*, 35-55.

[113] Midtgaard, J., Brabrand, C., & Wasowski, A. (2014). Systematic derivation of static analyses for software product lines. In *Proceedings of the 13th international conference on Modularity* (pp. 181-192). ACM.

[114] Brabrand, C., Ribeiro, M., Tolêdo, T., Winther, J., & Borba, P. (2013). Intraprocedural dataflow analysis for software product lines. In *Transactions on Aspect-Oriented Software Development X* (pp. 73-108). Springer, Berlin, Heidelberg.

[115] Gruler, A., Leucker, M., & Scheidemann, K. (2008). Modeling and model checking software product lines. In *International Conference on Formal Methods for Open Object-Based Distributed Systems* (pp. 113-131). Springer, Berlin, Heidelberg.

[116] Ter Beek, M. H., Lafuente, A. L., & Petrocchi, M. (2013). Combining declarative and procedural views in the specification and analysis of product families. In *Proceedings of the 17th International Software Product Line Conference co-located workshops* (pp. 10-17). ACM.

[117] Classen, A., Heymans, P., Schobbens, P. Y., Legay, A., & Raskin, J. F. (2010). Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (pp. 335-344). ACM.

[118] Cordy, M., Schobbens, P. Y., Heymans, P., & Legay, A. (2013). Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 472-481). IEEE Press.

[119] Cordy, M., Classen, A., Schobbens, P. Y., Heymans, P., & Legay, A. (2012). Managing evolution in software product lines: A model-checking perspective. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems* (pp. 183-191). ACM.

[120] Bettini, L., Damiani, F., & Schaefer, I. (2013). Compositional type checking of delta-oriented software product lines. *Acta Informatica*, *50*(2), 77-122.

[121] Liu, J., Basu, S., & Lutz, R. R. (2011). Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, *18*(1), 39-76.

[122] Garcés, K., Vara, J. M., Jouault, F., & Marcos, E. (2014). Adapting transformations to metamodel changes via external transformation composition. *Software & Systems Modeling*, *13*(2), 789-806.

[123] Favre, J. M. (2003). Meta-model and model co-evolution within the 3D software space. *ELISA*, *3*, 98-109.

[124] Hößler, J., Soden, M., & Eichler, H. (2005). Coevolution of models, metamodels and transformations. *Models and Human Reasoning*, (pp.129–154).

[125] Rose, L. M., Paige, R. F., Kolovos, D. S., & Polack, F. A. (2009). An analysis of approaches to model migration. In *Proc. Joint MoDSE-MCCM Workshop* (pp. 6-15).

[126] Narayanan, A., Levendovszky, T., Balasubramanian, D., & Karsai, G. (2009). Automatic domain model migration to manage metamodel evolution. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 706-711). Springer, Berlin, Heidelberg.

[127] Rose, L. M., Kolovos, D. S., Paige, R. F., & Polack, F. A. (2010). Model migration with epsilon flock. In *International Conference on Theory and Practice of Model Transformations* (pp. 184-198). Springer, Berlin, Heidelberg.

[128] Meyers, B., Wimmer, M., Cicchetti, A., & Sprinkle, J. (2012). A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST*, *42*.

[129] Garcés, K., Jouault, F., Cointe, P., & Bézivin, J. (2009). Managing model adaptation by precise detection of metamodel changes. In *European Conference on Model Driven Architecture-Foundations and Applications* (pp. 34-49). Springer, Berlin, Heidelberg.

[130] Herrmannsdoerfer, M., Benz, S., & Juergens, E. (2009). COPE-automating coupled evolution of metamodels and models. In *European Conference on Object-Oriented Programming* (pp. 52-76). Springer, Berlin, Heidelberg.

[131] Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *European Conference on Object-Oriented Programming* (pp. 600-624). Springer, Berlin, Heidelberg.

[132] Herrmannsdörfer, M., & Wachsmuth, G. (2014). Coupled evolution of software metamodels and models. In *Evolving Software Systems* (pp. 33-63). Springer, Berlin, Heidelberg.

[133] Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2008). Meta-model differences for supporting model co-evolution. In *Proceedings of the 2nd Workshop on Model-Driven Software Evolution-MODSE* (pp. 1–10).

[134] Gruschko, B. (2006). Towards structured revisions of metamodels and semi-automatic model migration. In *Position Paper for the Eclipse Modeling Symposium*.

[135] Sprinkle, J. M., & Karsai, G. (2003). *Metamodel driven model migration* (Doctoral dissertation), Vanderbilt University, USA.

[136] Sprinkle, J., & Karsai, G. (2004). A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, *15*(3-4), 291-307.

[137] Becker, S., Gruschko, B., Goldschmidt, T., & Koziolek, H. (2007). A process model and classification scheme for semi-automatic meta-model evolution. In *1st Workshop MDD, SOA und IT-Management (MSI), GI, GiTO-Verlag* (pp. 35-46).

[138] Gruschko, B., Kolovos, D., & Paige, R. (2007). Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution* (pp. 3-12). IEEE.

[139] Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference* (pp. 222-231). IEEE.

[140] Mantz, F., Taentzer, G., & Lamo, Y. (2013). Co-Transformation of Type and Instance Graphs Supporting Merging of Types and Retyping. *Electronic Communications of the EASST*, *61,* 1–24.

[141] Meyers, B., & Vangheluwe, H. (2011). A framework for evolution of modelling languages. *Science of Computer Programming*, *76*(12), 1223-1246.

[142] Meyers, B., Wimmer, M., Cicchetti, A., & Sprinkle, J. (2012). A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST*, *42(3),* 19-32.

[143] Wimmer, M., Kusel, A., Schönböck, J., Retschitzegger, W., Schwinger, W., & Kappel, G. (2010). On using inplace transformations for model co-evolution. In *Proc. 2nd Int. Workshop Model Transformation with ATL* (Vol. 711, pp. 65-78).

[144] Boussaïd, I., Siarry, P., & Ahmed-Nacer, M. (2017). A survey on search-based model-driven engineering. *Automated Software Engineering*, *24*(2), 233-294.

[145] Williams, J. R., Paige, R. F., & Polack, F. A. (2012). Searching for model migration strategies. In *Proceedings of the 6th International Workshop on Models and Evolution* (pp. 39-44). ACM.

[146] Kessentini, W., Sahraoui, H., & Wimmer, M. (2016). Automated metamodel/model co-evolution using a multi-objective optimization approach. In *European Conference on Modelling Foundations and Applications* (pp. 138-155). Springer, Cham.

[147] Kruse, S. (2011). On the use of operators for the co-evolution of metamodels and transformations. In *Proceedings of the International Workshop on Models and Evolution (ME 2011*), (pp. 54-63).

[148] Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schoenboeck, J., Schwinger, W., & Wimmer, M. (2015). Systematic co-evolution of OCL expressions. In *Proceedings of the 11th Asia-Pacific Conference on Conceptual Modelling*. (pp. 33-42). ACS.

[149] Levendovszky, T., Balasubramanian, D., Narayanan, A., & Karsai, G. (2009). A novel approach to semi-automated evolution of DSML model transformation. In *International Conference on Software Language Engineering* (pp. 23-41). Springer, Berlin, Heidelberg.

[150] Mendez, D., Etien, A., Muller, A., & Casallas, R. (2010). Towards transformation migration after metamodel evolution. In *International Workshop on Models and Evolution (ME2010)*.

[151] García, J., Diaz, O., & Azanza, M. (2012). Model transformation co-evolution: A semi-automatic approach. In *International Conference on Software Language Engineering* (pp. 144-163). Springer, Berlin, Heidelberg.

[152] Di Ruscio, D., Iovino, L., & Pierantonio, A. (2013). A methodological approach for the coupled evolution of metamodels and ATL transformations. In *International Conference on Theory and Practice of Model Transformations* (Vol. 7909, pp. 60-75), LNCS. Springer, Berlin, Heidelberg.

[153] Roser, S., & Bauer, B. (2008). Automatic generation and evolution of model transformations using ontology engineering space. In *Journal on Data Semantics XI* (pp. 32-64). Springer Berlin Heidelberg.

[154] Cabot, J., & Conesa, J. (2004). Automatic integrity constraint evolution due to model subtract operations. In *International Conference on Conceptual Modeling* (pp. 350-362). Springer, Berlin, Heidelberg.

[155] Kusel, A., Etzlstorfer, J., Kapsammer, E., Langer, P., Retschitzegger, W., Schoenboeck, J., Schwinger, W., Wimmer, M. (2014) A Systematic Taxonomy of Metamodel Evolution Impacts on OCL Expressions. *In Models and Evolution Workshop (ME 2014)*, CEUR-WS, vol. 1331, (pp. 2–11).

[156] Khelladi, D. E., Hebig, R., Bendraou, R., Robin, J., & Gervais, M. P. (2016). Metamodel and constraints co-evolution: A semi-automatic maintenance of OCL constraints. In *International Conference on Software Reuse* (pp. 333-349). Springer, Cham.

[157] Demuth, A., Lopez-Herrejon, R. E., & Egyed, A. (2013). Supporting the co-evolution of metamodels and constraints through incremental constraint management. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 287-303). Springer, Berlin, Heidelberg.

[158] Demuth, A., Lopez-Herrejon, R. E., & Egyed, A. (2012). Automatically generating and adapting model constraints to support co-evolution of design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (pp. 302-305). ACM.

[159] Marković, S., & Baar, T. (2005). Refactoring OCL annotated UML class diagrams. In *International Conference On Model Driven Engineering Languages And Systems* (pp. 280-294). Springer, Berlin, Heidelberg.

[160] Hassam, K., Sadou, S., Le Gloahec, V., & Fleurquin, R. (2011). Assistance system for OCL constraints adaptation during metamodel evolution. In *2011 15th European Conference on Software Maintenance and Reengineering* (pp. 151-160). IEEE.

[161] Object Management Group (OMG), *Unified Modeling Language (UML)*, Version 2.5.1. Needham, MA, 2017.

[162] Langer, P., Wieland, K., Wimmer, M., & Cabot, J. (2012). EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, *11*(1), 1-29.

[163] Kolovos, D. S., Rose, L. M., Matragkas, N. D., Paige, R. F., Polack, F. A., & Fernandes, K. J. (2010). Constructing and navigating non-invasive model decorations. In *International Conference on Theory and Practice of Model Transformations* (pp. 138-152). Springer, Berlin, Heidelberg.

[164] Alanen, M., & Porres, I. (2005). Version control of software models. In *Advances in UML and XML-based Software Evolution* (pp. 47-70). IGI Global.

[165] Oliveira, H., Murta, L., & Werner, C. (2005). Odyssey-VCS: a flexible version control system for UML model elements. In *Proceedings of the 12th international workshop on Software configuration management* (pp. 1-16). ACM.

[166] Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., & Helming, J. (2010). Operation-based conflict detection. In *Proceedings of the 1st International Workshop on Model Comparison in Practice* (pp. 21-30). ACM.

[167] Syriani, E., Gray, J., & Vangheluwe, H. (2013). Modeling a model transformation language. In *Domain Engineering* (pp. 211-237). Springer Berlin Heidelberg.

[168] Ramos, R., Barais, O., & Jézéquel, J. M. (2007). Matching model-snippets. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 121-135). Springer, Berlin, Heidelberg.

[169] Kramer, M. E. (2012). *Generic and extensible model weaving and its application to building models* (Master Thesis). Karlsruhe Institute of Technology.

[170] Sen, S., Mottu, J. M., Tisi, M., & Cabot, J. (2012). Using models of partial knowledge to test model transformations. In *International Conference on Theory and Practice of Model Transformations* (pp. 24-39). Springer, Berlin, Heidelberg.

[171] Sen, S., Moha, N., Baudry, B., & Jézéquel, J. M. (2009). Meta-model pruning. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 32-46). Springer, Berlin, Heidelberg.

[172] Fleurey, F., Steel, J., & Baudry, B. (2004). Validation in model-driven engineering: testing model transformations. In *Proceedings. 2004 First International Workshop on Model, Design and Validation, 2004.* (pp. 29-40). IEEE.

[173] Semeráth, O., Vörös, A., & Varró, D. (2016). Iterative and incremental model generation by logic solvers. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 87-103). Springer, Berlin, Heidelberg.

[174] Famelis, M., Ben-David, S., Chechik, M., & Salay, R. (2011). Partial models: A position paper. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation* (p. 1). ACM.

[175] Famelis, M., Salay, R., & Chechik, M. (2012). Partial models: Towards modeling and reasoning with uncertainty. In *2012 34th International Conference on Software Engineering (ICSE)* (pp. 573-583). IEEE.

[176] Eramo, R., Pierantonio, A., & Rosa, G. (2015). Managing uncertainty in bidirectional model transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (pp. 49-58). ACM.

[177] Li, X. (1999). A survey of schema evolution in object-oriented databases. In *Proceedings Technology of Object-Oriented Languages and Systems (Cat. No. PR00393)* (pp. 362-371). IEEE.

[178] Rahm, E., & Bernstein, P. (2006). An online bibliography on schema evolution. *ACM Sigmod Record* (Vol. 35, No. 4, pp. 30–31). ACM.

[179] Banerjee, J., Kim, W., Kim, H. J., & Korth, H. F. (1987). *Semantics and implementation of schema evolution in object-oriented databases* (Vol. 16, No. 3, pp. 311-322). ACM.

[180] Rahm, E., & Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *the VLDB Journal*, *10*(4), 334-350.

[181] Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, *37*(7), 383-393.

[182] Elsner, C., Botterweck, G., Lohmann, D., & Schroder-Preikschat, W. (2010). Variability in time—product line variability and evolution revisited. In *Proceedings of the Fourth International Workshop on Variability Modeling of Software-Intensive Systems*, (pp.131–137), ACM.

[183] Schubanz, M., Pleuss, A., Botterweck, G., & Lewerentz, C. (2012). Modeling rationale over time to support product line evolution planning. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems* (pp. 193-199). ACM.

[184] Seidl, C., & Aßmann, U. (2013). Towards modeling and analyzing variability in evolving software ecosystems. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM.

[185] Alwidian, S., & Amyot, D. (2017). Relaxing Metamodels for Model Family Support. In *11th Workshop on Models and Evolution (ME 2017)*. CEUR-WS, (Vol. 2019, pp. 64–70).

[186] Alwidian, S. (2017). Modeling Language Evolution for Model Family Support. In *MODELS (Satellite Events)*. CEUR-WS, (Vol. 2019, pp. 492-495).

[187] Jaoua A., Beaulieu, J.-M., Belkhiter, M., Deshernais, J., & Reguig, M. (1992) Optimal rectangular decomposition of a finite binary relation. In *Sixth Conference on International Conference on Discrete Mathematics*, University of Mysore, India.

[188] Jaoua A., Duwairi R., Elloumi S., & Yahia S.B. (2009) Data Mining, Reasoning and Incremental Information Retrieval through Non Enlargeable Rectangular Relation Coverage. In *Relations and Kleene Algebra in Computer Science. RelMiCS 2009*. LNCS, vol 5827 (pp. 199-210). Springer.

[189] Alwidian, S., & Amyot, D. (2019). Union Models: Support for Efficient Reasoning About Model Families Over Space and Time. In *International Conference on System Analysis and Modeling* (pp. 200-218). Springer, Cham.

[190] Mantz, F. (2014). *Coupled transformations of graph structures applied to model migration*. (Doctoral dissertation), Philipps-Universität Marburg, Germany.

[191] Taentzer, G., Prange, U., Ehrig, K., & Ehrig, H. (2006). *Fundamentals of Algebraic Graph Transformation*. EATCS. Springer.

[192] Taentzer, G., & Rensink, A. (2005). Ensuring structural constraints in graph-based models with type inheritance. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 64-79). Springer, Berlin, Heidelberg.

[193] Heckel, R., & Wagner, A. (1995). Ensuring consistency of conditional graph grammars-a constructive approach. *Electronic Notes in Theoretical Computer Science*, *2*, 118-126.

[194] Orejas, F., & Lambers, L. (2010). Symbolic attributed graphs for attributed graph transformation. *Electronic Communications of the EASST*, *30*:1–25.

[195] Rutle, A. (2010). *Diagram predicate framework: a formal approach to MDE*. (Doctoral dissertation), University of Bergen, Norway.

[196] Biermann, E., Ermel, C., & Taentzer, G. (2012). Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, *11*(2), 227-250.

[197] Ehrig, K., Küster, J. M., & Taentzer, G. (2009). Generating instance models from meta models. *Software & Systems Modeling*, *8*(4), 479-500.

[198] Mens, T. (2005). On the use of graph transformations for model refactoring. In *International Summer School on Generative and Transformational Techniques in Software Engineering* (pp. 219-257). Springer, Berlin, Heidelberg.

[199] de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2007). Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, *376*(3), 139-163.

[200] Ehrig, H., Prange, U., & Taentzer, G. (2004). Fundamental theory for typed attributed graph transformation. In *International conference on graph transformation* (pp. 161-177). Springer, Berlin, Heidelberg.

[201] Orejas, F., Ehrig, H., & Prange, U. (2010). Reasoning with graph constraints. *Formal Aspects of Computing*, *22*(3-4), 385-422.

[202] Kleppe, A., & Rensink, A. (2008). On a graph-based semantics for UML class and object diagrams. *Electronic Communications of the EASST*, *10*, 1-16.

[203] van der Straeten, R., Mens, T., Simmonds, J., & Jonckers, V. (2003). Using description logic to maintain consistency between UML models. In *International Conference on the Unified Modeling Language* (pp. 326-340). Springer, Berlin, Heidelberg.

[204] Heradio-Gil, R., Fernandez-Amoros, D., Cerrada, J.A. and Cerrada, C. (2011). Supporting commonality-based analysis of software product lines. *IET software*, *5*(6), 496-509.

[205] Famelis, M. (2016) *Managing design-time uncertainty in software models*. (Doctoral dissertation), University of Toronto, Canada.

[206] NetworkX, https://networkx.github.io/, last accessed 2019/06/05.

[207] SATisPY Solver, https://github.com/netom/satispy, last accessed 2019/06/15.

[208] Grubb, A. M., & Chechik, M. (2017). Modeling and Reasoning with Changing Intentions: An Experiment. In *2017 IEEE 25th International Requirements Engineering Conference (RE)* (pp. 164-173). IEEE.

[209] Fan, Y., Anda, A. A., & Amyot, D. (2018). An Arithmetic Semantics for GRL Goal Models with Function Generation. In *International Conference on System Analysis and Modeling* (pp. 144-162). Springer, Cham.

[210] Yu, E. S. (1997). Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of ISRE'97: 3rd IEEE International Symposium on Requirements Engineering* (pp. 226-235). IEEE.

[211] Chung, L., Nixon, B., Yu, E., & Mylopoulos, J. (2000). Non-functional requirements in software engineering. Kluwer Academic Publishers. *Massachusetts, USA*.

[212] Giorgini, P., Mylopoulos, J., & Sebastiani, R. (2005). Goal-oriented requirements analysis and reasoning in the Tropos methodology. *Engineering Applications of Artificial Intelligence*, *18*(2), 159-171.

[213] van Lamsweerde, A. (2009). *Requirements engineering: From system goals to UML models to software* (Vol. 10). Chichester, UK: John Wiley & Sons.

[214] Amyot, D., Ghanavati, S., Horkoff, J., Mussbacher, G., Peyton, L., & Yu, E. (2010). Evaluating goal models within the goal-oriented requirement language. *International Journal of Intelligent Systems*, *25*(8), 841-877.

[215] Horkoff, J., & Yu, E. (2010). Finding solutions in goal models: an interactive backward reasoning approach. In *International Conference on Conceptual Modeling* (pp. 59-75). Springer, Berlin, Heidelberg.

[216] Sebastiani, R., Giorgini, P., & Mylopoulos, J. (2004). Simple and minimum-cost satisfiability for goal models. In *International Conference on Advanced Information Systems Engineering* (pp. 20-35). Springer, Berlin, Heidelberg.

[217] Sun, W., & Yuan, Y. X. (2006). *Optimization theory and methods: nonlinear programming* (Vol. 1). Springer Science & Business Media.

[218] Anda, A. A. & Amyot, D. (2019). Arithmetic Semantics of Feature and Goal Models for Adaptive Cyber-Physical Systems. In *27th IEEE International Requirements Engineering Conference (RE'19)* (pp. 245-256). IEEE CS.

[219] IBM: *IBM ILOG CPLEX Optimization Studio* (2019), https://www.ibm.com/ca-en/marketplace/ibm-ilog-cplex.

[220] Rossi, F., Van Beek, P., & Walsh, T. (Eds.). (2006). *Handbook of constraint programming*. Elsevier.

[221] Mussbacher, G., Ghanavati, S., & Amyot, D. (2009). Modeling and Analysis of URN Goals and Scenarios with jUCMNav. In *2009 17th IEEE International Requirements Engineering Conference* (pp. 383-384). IEEE CS.

[222] *jUCMNav*, Version 7.0.0, University of Ottawa Online: http://softwareengineering.ca/jucmnav (accessed October 2019)

[223] SymPy. https://www.sympy.org/en/index.html [online: accessed October, 2019].

[224] Fan, Y.: *GRLToMath plugin for jUCMNav* (2018). https://github.com/AAmber-Fan/GRLToMath.

[225] Alwidian, S. & Amyot, D. (2019). Inferring Metamodel Relaxations based on Structural Patterns to Support Model Families. In *13th Workshop on Models and Evolution (ME 2019)*. IEEE CS.

[226] Cho, H., & Gray, J. (2011). Design patterns for metamodels. In *SPLASH Workshops*, (pp. 25–32). ACM.

[227] Kitchenham, B., Linkman, S., & Law, D. (1997). DESMET: a methodology for evaluating software engineering methods and tools. *Computing & Control Engineering Journal*, *8*(3), 120-126.

[228] CI&T, *The Four Types of Analytics*. http://www.ciandt.com/card/four-types-of-analytics-and-cognition [online; accessed: Dec 2, 2017].

[229] Perry, D. E., Porter, A. A., & Votta, L. G. (2000). Empirical studies of software engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering* (pp. 345-355). ACM.

Appendix A: GRL Metamodel and Syntax

Figure 101 provides an extract of the GRL metamodel, and Figure 102 illustrates the basic elements of GRL models.

Figure 101 Extract of the GRL metamodel (ITU-T [6])

GRL Model Elements

A GRL model consists of the following elements:

- Intentional elements:
  1) Softgoal ( ) is an intention, related to NFRs, that has no clear objective measure of satisfaction.
  2) Goal ( ) is an intention, related to functional requirements, that can be fully satisfied in a quantifiable way.

3) Task (⬭) represents activities or solutions used to meet goals and softgoals.
4) Resource (▭) is needed in order to achieve softgoals, goals, and tasks.

- Intentional Links:
  1) Decomposition links (+—) are used to connect elements and sub elements using AND, OR and XOR decomposition relationships.
  2) Contribution links (⟶) indicate the positive or negative impact of the satisfaction of one element on the satisfaction of another element.
  3) Dependency links (-●-) model dependencies between elements, often across two different actors, to show that an actor depends on another actor to meet a goal, satisfy a softgoal, perform a task, provide a resource, etc.
- Actors (⬭): are the active entities in the system (e.g., stakeholders or systems themselves) who want goals to be achieved, softgoals to be satisfied, tasks to be performed, and resources to be available.

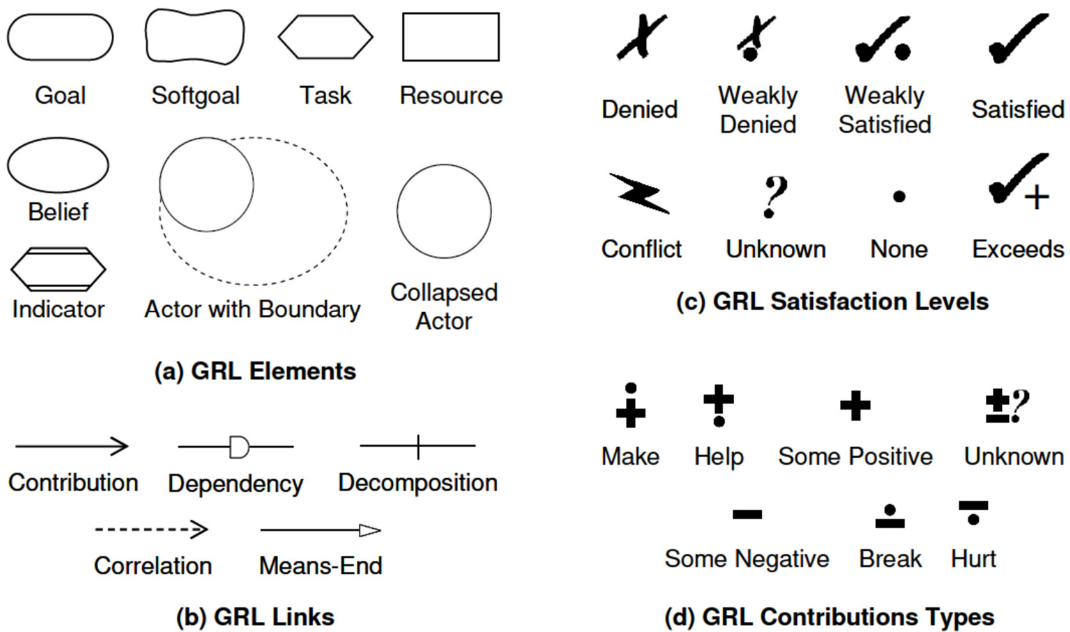


Figure 102 Summary of the GRL graphical syntax

GRL Evaluation Strategies

A GRL *strategy* describes a particular configuration of alternatives in the GRL model by assigning an initial qualitative or quantitative evaluation value (an integer value between 100 and 100) to some of the intentional elements in the model. A GRL *evaluation mechanism* then propagates evaluation values to other high-level intentional elements, through intentional links, and compute their satisfaction values. Strategies can be compared with each other to help decide on the most appropriate trade-offs among conflicting goals of stakeholders and to choose the best alternative that satisfy these goals. In addition, GRL supports an *importance* attribute for intentional elements inside actors (also quantitative or qualitative). This attribute is considered when evaluating strategies for the goal model, resulting in satisfaction levels measured at the actor level [7].

Appendix B: Summary of STAL's Syntax

This appendix provides the grammar of the Spatio-Temporal Annotation Language (STAL). In addition, it provides examples for all possible cases of annotations using this language, based on the MergeSTAL algorithm provided in Listing 1, Section 3.4.3.

Metagrammar

The following symbols are used to define the grammar:

- <> for rules
- ::= for definition
- { }* for 0 to many
- { }+ for 1 to many
- | for alternatives
- **bold** for terminal symbols
- NATURAL for non-negative integers
- IDENTIFIER for Strings without spaces
- *# text* for comments

STAL Grammar

```
# ALL here means all versions and configurations.
<STAL> ::= <annotation> {; <annotation>}* | ALL
<annotation> ::= < <versions> , <configurations> >
# ALL here means all versions.
<versions> ::= <singleversion> | <listversions> | <rangeversions>
                | ALL
# In versions, ver1, ver2, ver3, … are sorted.
<singleversion> ::= verNATURAL
```

```
# Nested lists, if any, are flattened.
<listversions> ::= ( <versions> {, <versions>}+ )
# The first version value must be lower than the second version value.
<rangeversions> :: = [ <singleversion> : <singleversion> ]
# ALL here means all configurations.
<configurations> ::= <singleconfig> | <listconfigs> | ALL
<singleconfig> ::= IDENTIFIER
<listconfigs> ::= ( <singleconfig> {, <singleconfig>}+ )
```

Listing 6 STAL grammar

Note that some modelers might use a hierarchical version numbering scheme (versions 2.3.1 and 4.3.2, etc.). However, these hierarchical numbers can be mapped to simple integers (while keeping the ordering) for simplification.

Illustrative Examples of Annotating Elements in one M_U

In the next five examples, we assume that there is a model family *MF* consisting of sixteen versions and three configurations of a model M, and that their union is captured by M_U.

Example 1: An element that belongs to <ver_1, $conf_A$>, <ver_2, $conf_A$>, <ver_3, $conf_A$>, <ver_1, $conf_B$>, and <ver_4, $conf_B$> will be annotated in M_U as: <[ver_1:ver_3], ($conf_A$)>; <(ver_1,ver_4), ($conf_B$)>

Example 2: An element that belongs to <ver_1, $conf_A$>, <ver_2, $conf_A$>, <ver_3, $conf_A$>, <ver_1, $conf_B$>, <ver_2, $conf_B$>, <ver_3, $conf_B$>, <ver_4, $conf_B$> will be annotated in M_U as: <[ver_1:ver_3], ($conf_A$, $conf_B$)>; <(ver_4), ($conf_B$)>

Example 3: An element that belongs to <ver_1, $conf_A$>, <ver_2, $conf_A$>, <ver_3, $conf_A$>, <ver_5, $conf_A$>, <ver_6, $conf_A$>, <ver_7, $conf_A$>, and also to <ver_1, $conf_B$>, <ver_2, $conf_B$>, <ver_3, $conf_B$>, <ver_5, $conf_B$>, <ver_6, $conf_B$>, <ver_7, $conf_B$> will be annotated in M_U as: <([ver_1:ver_3],[ver_5:ver_7]),($conf_A$,$conf_B$)>

Example 4: An element that belongs to <ver_1, $conf_A$>, <ver_2, $conf_A$>, <ver_3, $conf_A$>, <ver_5, $conf_A$>, <ver_7, $conf_A$>, <ver_{10}, $conf_A$>, and also to <ver_1, $conf_B$>, <ver_2, $conf_B$>, <ver_3, $conf_B$>, <ver_5, $conf_B$>, <ver_7, $conf_B$>, <ver_{10}, $conf_B$> will be annotated in M_U as: <([ver_1:ver_3],ver_5,ver_7,ver_{10}),($conf_A$, $conf_B$)>

Example 5: An element that belongs to <ver_1, $conf_A$>, <ver_2, $conf_A$>, <ver_3, $conf_A$>, <ver_7, $conf_A$>, <ver_8, $conf_A$>, <ver_9, $conf_A$>, <ver_{11}, $conf_A$>, <ver_{16}, $conf_A$>, and to <ver_1, $conf_B$>, <ver_2, $conf_B$>, <ver_3, $conf_B$>, <ver_7, $conf_B$>, <ver_8, $conf_B$>, <ver_9, $conf_B$>, <ver_{11}, $conf_B$>, <ver_{16}, $conf_B$>, and also to <ver_4,$conf_C$> will be annotated in M_U as: <([ver_1:ver_3],[ver_7:ver_9],ver_{11},ver_{16}), ($conf_A$, $conf_B$)>; <(ver_4), ($conf_C$)>

Illustrative Examples of Annotating Elements in M_{Unew}

In the next two examples, we assume that M_U is already constructed for the same *MF* discussed above (i.e., *MF* consists of 16 versions and 3 configurations), and that a model M17 is added to the family, with all of M17's elements annotated as <ver_{17}, $conf_A$>.

Example 6: An element that belongs to M_U and annotated with <[ver_1:ver_3], ($conf_A$)>; <(ver_1,ver_4), $conf_B$> will be annotated in $M_{Unew}= M_U \cup M_{17}$ as: <([ver_1:ver_3],ver_{17}), ($conf_A$)>; <(ver_1, ver_4), $conf_B$>

Example 7: An element that belongs to M_U and annotated with <ALL>, where *ALL* corresponds to <[ver_1:ver_{16}], ($conf_A$, $conf_B$, $conf_C$)>, will be annotated in $M_{Unew}= M_U \cup M_{17}$ as: <[ver_1:ver_{16}], ($conf_A$, $conf_B$, $conf_C$)>; <ver_{17}, $conf_A$>.

Example 8: If a model M17 is added to the family, with all of M17's elements annotated as <ver_{17}, $conf_A$, $conf_B$, $conf_C$ >, then an element in M_U that is annotated with <ALL> will be annotated in $M_{Unew}= M_U \cup M_{17}$ as <ALL>.

Example 9: The same principle discussed in examples 6-8 applies also when merging two union models together, such that $M_{Unew} = M_{U1} \cup M_{U2.}$ If an element in M_{U1} annotated with <[ver_1:ver_6], ($conf_A$, $conf_B$)>; <ver_9, $conf_A$> is unified with the same element from M_{U2} annotated with <(ver_8, ver_{11}, ver_{13}), ($conf_A$, $conf_B$, $conf_C$)>, then that element will be annotated in M_{Unew} as: <[ver_1:ver_6], ($conf_A$, $conf_B$)>; <ver_9, $conf_A$>; <(ver_8, ver_{11}, ver_{13}), ($conf_A$, $conf_B$, $conf_C$)>

Appendix C: Sample CPLEX Code

This appendix provides CPLEX code for four GRL models of the model family in Chapter 6 (Figure 70), and also the CPLEX code of their union model. Other models are available online at http://bit.ly/2BW1zIG.

For brevity, goals, softgoals, and tasks are renamed as N1, N2, N3, etc. Also, in this example, we searched for a solution that maximizes the satisfaction value of the root goal "Manage Home", renamed as N1.

N1	ManageHome
N2	ProtectHome
N3	RefreshAirInside
N4	GiveIllusionHouseLivedIn
N5	PreventThiefFromEntering
N6	OpenWindow
N7	TurnVentilatorOn
N8	TurnLightOnAndOff
N9	TurnOnMusic
N10	LookDoorsAndWindows
N11	EnergySavedWisely
N12	IncreasedPrivacy
N13	OpenWindow
N14	TurnVentilatorOn
N15	TurnLightOnAlways
N16	CalmSenioDown

```
//Student smart home, Version 1 (i.e., ver1,ConfA)
using CP;
dvar int N6 in 0..100;
dvar int N7 in 0..100;
dvar int N8 in 0..100;
dvar int N9 in 0..100;
dvar int N10 in 0..100;
dexpr float N3 = ( maxl(0, minl(100,((N6*75) +(N7*25))/100)));
dexpr float N4 = ( maxl(0, minl(100,((N8*50) +(N9*50))/100)));
dexpr float N5 = ( maxl(0, minl(100,((N10*100))/100)));
dexpr float N2 = ( minl(N4, N5) );
dexpr float N1 = ( minl(N2, N3) );
dexpr float N12 = ( maxl(0, minl(100,((N10*100) +(N6*-75) +(N7*60))/100)));
dexpr float N11 = ( maxl(0, minl(100,((N6*90) +(N7*-30))/100)));
maximize N1;
```

```
///////////// Solution for Student smart home (ver1,ConfA)////////////////
// solution for Mod1A with objective 100
N8 = 100;
N9 = 100;
N10 = 100;
N6 = 100;
N7 = 100;
////////////////////////////////////////////////////////////////////////////
```

```
// Student smart home, Version 2 (i.e., ver2,ConfA),
using CP;
dvar int N6 in 0..100;
dvar int N7 in 0..100;
dvar int N8 in 0..100;
dvar int N9 in 0..100;
dvar int N10 in 0..100;
dexpr float N3 = ( maxl(0, minl(100,((N6*25) +(N7*75))/100)));
dexpr float N4 = ( maxl(0, minl(100,((N8*50) +(N9*50))/100)));
dexpr float N5 = ( maxl(0, minl(100,((N10*100))/100)));
dexpr float N2 = ( minl(N4, N5) );
dexpr float N1 = ( minl(N2, N3) );
dexpr float N12 = ( maxl(0, minl(100,((N10*100) +(N6*-30) +(N7*75))/100)));
dexpr float N11 = ( maxl(0, minl(100,((N6*60) +(N7*-75))/100)));
maximize N1;
///////////// Solution for Student smart home (ver2,ConfA)////////////////
// solution for Mod2A with objective 100
N8 = 100;
N9 = 100;
N10 = 100;
N6 = 100;
N7 = 100;
////////////////////////////////////////////////////////////////////////////
```

```
// Senior smart home, Version 1 (i.e., ver1,ConfB),
using CP;
dvar int N6 in 0..100;
dvar int N7 in 0..100;
dvar int N8 in 0..100;
dvar int N9 in 0..100;
dvar int N15 in 0..100;
dvar int N10 in 0..100;
dexpr float N3 = ( maxl(0, minl(100,((N6*50) +(N7*50))/100)));
dexpr float N4 = ( maxl(0, minl(100,((N8*33) +(N9*33) +(N15*33))/100)));
dexpr float N5 = ( maxl(0, minl(100,((N10*100))/100)));
dexpr float N2 = ( minl(N4, N5) );
dexpr float N1 = ( minl(N2, N3) );
dexpr float N12 = ( maxl(0, minl(100,((N10*100) +(N6*-50) +(N7*60))/100)));
dexpr float N11 = ( maxl(0, minl(100,((N6*50) +(N7*-75) +(N15*-75))/100)));
maximize N1;


///////////// Solution for Senior smart home (ver1,ConfB) ////////////////
```

```
// solution with objective 99
N8 = 100;
N9 = 100;
N15 = 100;
N10 = 100;
N6 = 100;
N7 = 100;
```

```
////////////////////////////////////////////////////////////////////////////
```

```
// Senior smart home, Version 2 (i.e., ver2,ConfB)
using CP;
dvar int N6 in 0..100;
dvar int N7 in 0..100;
dvar int N8 in 0..100;
dvar int N9 in 0..100;
dvar int N15 in 0..100;
dvar int N10 in 0..100.
dexpr float N3 = ( maxl(0, minl(100,((N6*50) +(N7*50))/100)));
dexpr float N4 = ( maxl(0, minl(100,((N8*33) +(N9*33) +(N15*33))/100)));
dexpr float N5 = ( maxl(0, minl(100,((N10*100))/100)));
dexpr float N2 = ( minl(N4, N5) );
dexpr float N1 = ( minl(N2, N3) );
dexpr float N12 = ( maxl(0, minl(100,((N10*100) +(N6*-50) +(N7*90))/100)));
dexpr float N11 = ( maxl(0, minl(100,((N6*50) +(N7*-75) +(N15*-75))/100)));
dexpr float N16 = ( maxl(0, minl(100,((N9*75))/100)));
maximize N1;

///////////////////Solution for Mod2B (Winter_Senior)//////////////////
// solution with objective 99
N8 = 100;
N9 = 100;
N15 = 100;
N10 = 100;
N6 = 100;
N7 = 100;
```

```
/////////////////////////////////////////////////////////////////////////////
```

```
//Union Model of the four models:
using CP;
dvar int ver in 1..2;
dvar int conf in 1..2; // conf==1 corresponds to ConfA and conf==2
                         // corresponds to ConfB.

dvar int N6 in 0..100;
dvar int N7 in 0..100;
dvar int N8 in 0..100;
dvar int N9 in 0..100;
dvar int N10 in 0..100;
dvar int N15 in 0..100;
dexpr float N3 = ((maxl(0, minl(100,(((ver==1 && conf==1 )? (N6*25) +(N7*75):
(ver==1 && conf==2 )? (N6*50) +(N7*50):
(ver==2 && conf==1 )? (N6*75) +(N7*25):
(ver==2 && conf==2 )? (N6*50) +(N7*50):
```

```
0 ))/100))));
dexpr float N5 = ((maxl(0, minl(100,(((N10*100)))/100))));
dexpr float N4 = ((maxl(0, minl(100,(((ver==1 && conf==1 )? (N8*50) +(N9*50):
(ver==1 && conf==2 )? (N8*33) +(N9*33) +(N15*33):
(ver==2 && conf==1 )? (N8*50) +(N9*50):
(ver==2 && conf==2 )? (N8*33) +(N9*33) +(N15*33):
0 ))/100))));
dexpr float N2 = (((minl(N4,N5)) ));
dexpr float N1 = (((minl(N2,N3)) ));
dexpr float N12 = ((maxl(0, minl(100,(((N10*100) +ver==1 && conf==1 ? (N6*-30)
+(N7*75):
(ver==1 && conf==2 )? (N6*-50) +(N7*60):
(ver==2 && conf==1 )? (N6*-75) +(N7*60):
(ver==2 && conf==2 )? (N6*-50) +(N7*90):
0 ))/100))));
dexpr float N11 = ((maxl(0, minl(100,(((ver==1 && conf==1 )? (N6*60) +(N7*-
75):
(ver==1 && conf==2 )? (N6*50) +(N7*-75) +(N15*-75):
(ver==2 && conf==1 )? (N6*90) +(N7*-30):
(ver==2 && conf==2 )? (N6*50) +(N7*-75) +(N15*-75):
0 ))/100))));
dexpr float N16 = ((maxl(0, minl(100,(((ver==2 && conf==2 )? (N9*75):
0 ))/100))));

maximize N1;

//////////////////Solution for Union Model MU ///////////
// solution with objective 100
ver = 1;
conf = 1;
N8 = 100;
N9 = 100;
N15 = 0;
N10 = 100;
N6 = 100;
N7 = 100;
```